TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Tarkvarateaduse instituut

Multithreaded computations of the Maximum Clique Algorithm based on Recoloring.

bakalaureusetöö

Üliõpilane: Viktor Viguro

Üliõpilaskood: 970800IAPB

Juhendaja: Deniss Kumlander, vanemteadur

Tallinn 2019

Autorideklaratsioon.

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Viktor Viguro

19.05.19

Bakalaureusetöö ülesandepüstitus.

Üliõpilane:Viktor Viguro (970800IAPB)

Teema: Paralleelne arvutamine suurima kliki leidmiseks.

Juhendaja: Deniss Kumlander, tarkvarateaduse instituudi vanem teadur

Probleem: Suurima kliki leidmine võtab palju aega ja ressursse, juhul kui graaf on päris suur. Isegi väike progressi selles küsimuses võib aidata NP-täielike ja NP-raske probleemide lahendamiseks.

Eesmärk: Kontrollida üle hüpoteesi: saab kiirendada suurima kliki leidmise protsessi, juhul kui kasutatakse mitme arvutuse voogu. Selleks on vaja teha kolm sammu: 1. Aru saada algoritmi; 2. Leida võimalikud paraleeliseerimis kohad; 3. Verifetseerida kas kõik neid soobivad selleks, et suurendada kiirus ning välja valida kõige optimaalsema varianti. Näidata probleeme ja/või piirangud mis tulevad teaduse protsessi ajal. Leida edu või ebaõnnestumise põhjused ja määrata suunad teadusuuringutele, et saab edasi liiguta.

Metoodika: Adapteerida ja/või muuta olemasoleva algoritmide programmi koodi, et saab kasutada mitme arvutuse voogu võimalused. Koostata testi, et määrata prograami töötamise kiiruse.

Alikad:

1. **“Some practical algorithms to solve the maximum clique problem”**, Deniss Kumlander, TTÜ, Doktoritöö, 2005a, ISBN 9985595815, ISSN 14064731
2. **“Pöördotsingu suurima kliki leidmise algoritm ülevärvimise teel. Reversed Search Maximum Clique Algorithm Based on Recoloring”**, Porošin, Aleksandr, TTÜ, Magistritöö, 2015 a.
3. **“Essential Algorithms”,** Rod Stephens, 2013, ISBN 1118612108.

Annotatsioon.

Maailmas on palju teaduslike küsimusi, millele pole vastuseid või ei saa nii lihtsalt leida lahendust, isegi kui on olemas kõige võimsam arvuti.

Palju probleeme saab parandada ainult algoritmi kasutamisel. Aga on terve probleemide rida, mida ei lahenda ainult algoritmi abil.

Nende probleemide vastuste leidmine on üliraske ja võtab palju aega ja tehnilisi ressursse, aga lahendust ei pruugi leida teadlase elu lõpuni. Need probleemid on tuntud nagu NP-täielikud probleemid.

Suurima kliki leidmine on üks enamlevinud NP-täielikest probleemidest.

Antud töö kontrollib üle hüpoteesi: kas saab kiirendada suurima kliki leidmise protsessi, juhul kui kasutatakse mitme arvutite voogu (threads). Kui saab, siis määrame need osad, mis annavad suurima ajavõidu.

Põhiidee on leida algorütmid, millega saame suurima ajavõidu, kui kasutame mitme arvutite voogu (threads). Teeme samad algorütmid mitme tööjaamadega ja näitame tulemust. Kas saab arvutusressurside suurendamisega kliki leidmise aja vähendada ja kuidas seda tulemust saab teha paremaks.

Lõpuks võrdleme saadud tulemuste tabeli ja graafiku abil.

Eesmärk on veel ühe sammu tegemine NP-täielike probleemide lahendamises. See saab ka aidata teise NP-täielike ja NP-raske probleemide lahendamise leidmiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 109 leheküljel, 6 peatükki, 47 joonist, 6 tabelit.

Abstract.

There are many scientific problems in the world that have no answers or solutions not exist for now. Even if we have the most powerful computers.

Many problems can be solved only by using the original algorithm. But there is a whole series of problems that can not be solved by the original algorithm only.

Finding the answers to these problems is quite hard and takes a lot of time and technical resources. Finding a solution can take all the time until the end of the researcher's life. These problems are known as NP-complete problems.

Finding the maximum clique is one of the most common NP-complete problems.

This work verifies the hypothesis: could maximum clique finding process be accelerated by the threads of multiple counting are used. If so, let's determine the parts that give the greatest time.

The basic idea is to find the part of algorithms that we can get the most time wins with using threads of multiple computing. Let's do the same algorithms with multiple workstations and show the result. Can we decrease the time of finding maximum clique by increasing your computing resources and how we can make that result better?

Finally, we compare the results obtained with the tables and graphics.

The goal is to take one more step in solving NP-complete problems. It can also help in finding another NP-complete and NP-heavy problem solving.

The thesis is in English and contains 109 pages of text, 6 chapters, 47 figures, 6 tables.

Abbreviations Glossary

CBC Current best clique. Abbreviation used in multiple algorithms to define an array for storing the largest clique vertices found by far. Sometimes it is used as |CBC| that means the number of vertices contained in a current best clique.

DIMACS Center for Discrete Mathematics and Theoretical Computer Science. Presents a pack of benchmarks instances, which represent different graphs, constructed on the real life problem basis. These instances can be used for testing maximum clique algorithms performance.

NP complexity class Nondeterministic polynomial time complexity class. Class of problems that can be solved with a polynomial amount of time by a nondeterministic Turing machine.

P complexity class Polynomial time complexity class. Class of problems that can be solved with a polynomial amount of time by a deterministic Turing machine.

VRecolor-BT-u Vertex recolor with backtracking for unweighted cases. A new exact algorithm presented in the current thesis based on VColor-BT-u. The main idea is to apply additional in-depth coloring (recoloring) to fasten the maximum clique search published by Aleksandr Porošin [Porošin 2015].

VRecolor-BT-u-VP, VRecolor-BT-u-VP2, VRecolor-BT-u-VPtest
The same algorithm that VRecolor-BT-u, with added some parallel calculations in the code for testing hypotheses that multithreading calculations can make estimated time lower.

Table of Contents

List of figures

List of tables

1. Introduction.

1.1 Graph theory.

Graph is a structure what describes connections between two objects. This objects named “vertex” and vertex is connected to each other with edge (lines between dots (vertex)). Graph theory is part of discrete mathematics (Figure 1.0).

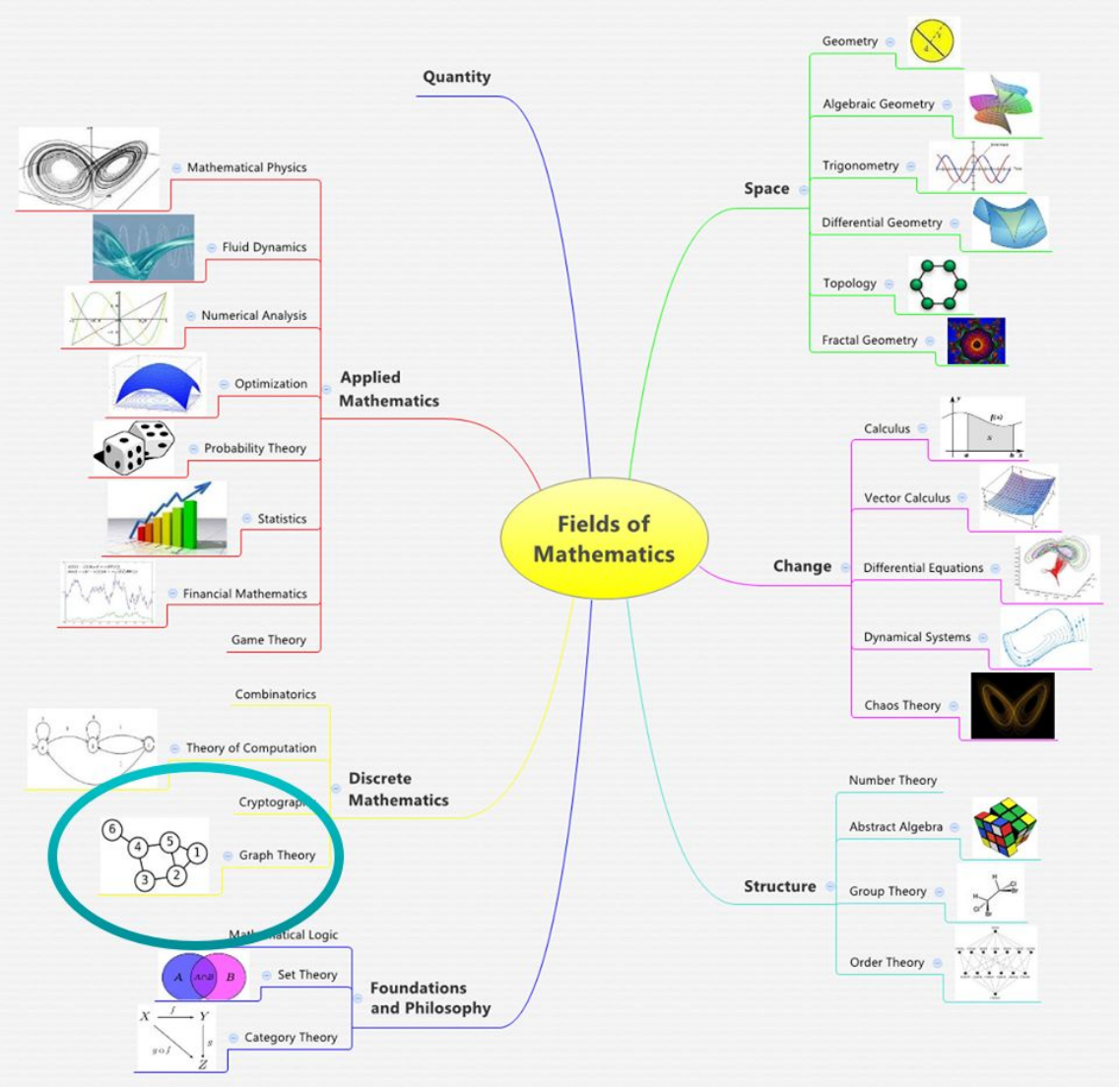


Figure 1.0 Graph theory place in mathematics.

Graph theory is a tool that can help to transform real life problem into special representation i.e. graphs. A lot of everyday problems can be solved with graph theory tools.

All of taxi apps like Bolt, Uber, Yandex.taxi with navigation used graph theory tools to connect drivers and clients. Graph theory made finding free taxi car easier and faster, helps to build optimal route to destination point.

With graph theory we can organize flight lines in the world. It makes possible to plan long distance trips automatically, find and calculate trip price in one mouse click.

Navigation that used graph theory can calculate optimal route even in rush hours.

Graph theory we use every day it is invisible for regular users, but without it all this services (navigation, flight and route planners and others) can't work fast and correctly.

Google (and Yandex) uses graph in they search engine. Each article (on every site that was put in index) have connections to other sites, article or citats. Site which have maximum citatast have better place in searching list. This model is provided by graph with connection. Each connection have weight. And vertex (site) which have maximum connections weight gets higher rating. Google changes rating algorithm time by time by adding more and more properties for graph vertex. And makes search results more relevant for final user.

For each user in real time makes individual searching model that depends on his personal interests and habits. Interest and habit model was made from users behavior in internet surfing.

Graph theory is used in:

- chemistry (to describe structures, paths of complex reaction) the phase rule can also be interpreted as a problem in graph theory;
- computer chemistry is a relatively young field of chemistry, based on the application of graph theory. Graph theory is the mathematical basis of chemoinformatics. Graph theory allows you to accurately determine the number of theoretically possible isomers of hydrocarbons and other organic compounds;
- computer science and programming (graph-diagram of the algorithm, automata);

- communication and transport systems. In particular, for routing data on the Internet;
- economics;
- logistics;
- circuitry (the topology of interconnections of elements on a printed circuit board or microcircuit is a graph or a hypergraph).

Graph theory main concepts began to use English scientist, mathematical J.Sylvester 1878a. J.Sylvester was the first scientist who began to use the “graph” word in his work.

First Graph theory problem present Leonhard Euler 1736a. Leonhard Euler (1707-1783) was a Swiss mathematician, physicist, astronomer, logician, and engineer.

In 1735, Euler presented a solution to the problem known as the Seven Bridges of Königsberg [1]. The city of Königsberg, Prussia was set on the Pregel River and included two large islands that were connected to each other and the mainland by seven bridges. The problem is to decide whether it is possible to follow a path that crosses each bridge exactly once and returns to the starting point.

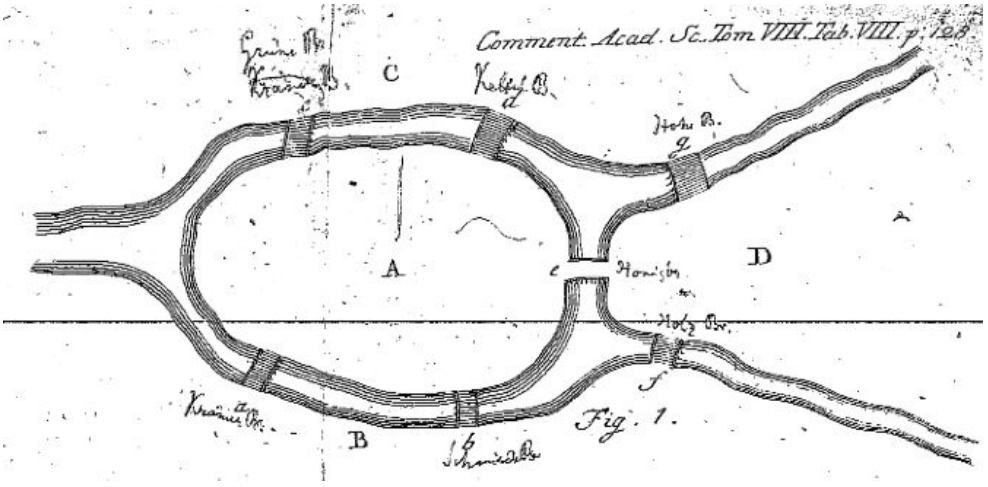


Figure 1.1 The Königsberg Bridge Problem. Map representation [F1]

Figure 1.2 The Königsberg Bridge Problem. Map to a graph representation [F2]

Leonhard Euler (1707-1783) solved the Königsberg Bridge Problem. He proved that it is impossible, in other words “there is no Eulerian circuit”.

Graphs can be used to model many types of relations and processes in physical, biological, social and information systems. Nowadays with graph theory solving problems in Computer science, Linguistics, Physics and chemistry, Social sciences, Biology, Mathematics and other topics (for example travel/flight time and cost).

1.2 Graph theory base.

Graaf G(V, E), where V={ V1,V2 , ….., Vn } is vertex and E={ E1, E2, ….., Em } edge, which connects vertex for each other.

G(Ei)={ Ek | < Ei , Ek> ∈V}

Below are presents four main ways to represent graphs G(V, E) (Figure 1.3 - 1.6).

	V1	V2	V3	V4	V5	V6
V1		E<1,2>			E<1,5>	
V2	E<1,2>		E<2,3>		E<2,5>	
V3		E<2,3>		E<3,4>		
V4			E<3,4>		E<4,5>	E<4,6>
V5	E<1,5>	E<2,5>		E<4,5>		
V6				E<4,6>		

Figure 1.3 Example of graph G (table view (2) of graph):

	V1	V2	V3	V4	V5	V6
V1	0	1	0	0	1	0
V2	1	0	1	0	1	0
V3	0	1	0	1	0	0
V4	0	0	1	0	1	1
V5	1	1	0	1	0	0
V6	0	0	0	1	0	0

Figure 1.4 Example of graph G (table view (2) of graph):

1,2; 1,5; 2,5; 2,3; 3,4; 4,5; 4,6

Figure 1.5 Example of graph G (listview of graph):

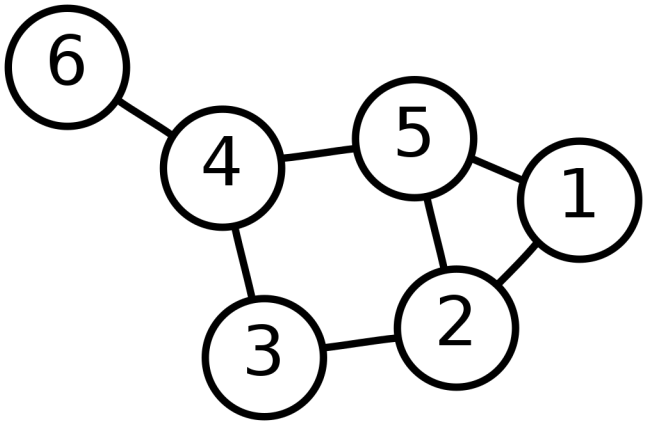


Figure 1.6 Example of graph G (picture view of graph):

Each connection (E, edge) in graph may have cost (weight) and direction depend on context of the solving problem. Weight is a number (non negative integer) assigned to each edge or vertex that can be represent additional property like flite cost, road length, reqwired power and etc.

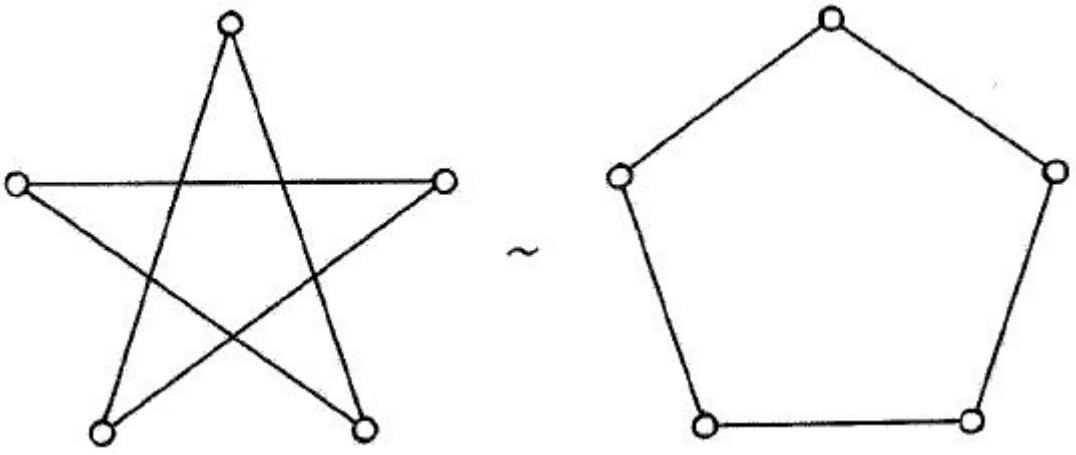


Figure 1.7 Two equivalent ways to draw graph.[4]

As you can see on Figure 1.7 one graph could be drawn in many ways. Both draws represent exactly the same set of edges and vertices, in other words represent the same graph.

As you can see on Figure 1.7 one graph could be drawn in many ways. Both draws represent exactly the same set of edges and vertices, in other words represent the same graph.

In fact, choosing a graph representation method is not so simple. It is necessary to clearly understand what tasks will be solved and which graph will be processed. Naturally, for a graph of 10 vertices there will be no difference. But on large graphs it is important and difference is noticeable.

The right choice is not always clear. There is a common misconception that an adjacency matrix requires a lot more memory than an adjacency list (it stores many empty values). But this is true only for loosely coupled graphs. If you work with a strongly connected graph, then the difference in memory consumption will not be so great, but the adjacency matrix will provide a faster search for edges.

There are many ways to represent graphs. But the most common are the adjacency matrix and adjacency list. If you do not know exactly what you need, use one that suits you better.

In this job as graph we understand undirected graph with have no loops. Loops there is edge which begins and ends in one vertex.

1.3 Complexity

Complexity is your enemy.
Any fool can make something complicated.
It is hard to make something simple.
(Richard Branson) [6]

1.3.1. Complexity classes, algorithm speed

Let we have algorithm which working time is X seconds. In Figure 1.8 we can see how algorithm speed in seconds to minutes, days, weeks, etc. In other words how long you need to wait calculating result in real life. If an algorithm working x100 times slower we must wait for example almost 4 month instead of 1.1 days (from $\mathbf{10}^5$ to $\mathbf{10}^7$ seconds).

Figure 1.8 shows how important it is to optimize algorithms a special complex and resource intensive algorithms.

X seconds (algorithm speed)	time (when the result is coming)
$\mathbf{10}^2$	1,7 minutes
$\mathbf{10}^4$	2,8 hours
$\mathbf{10}^5$	1,1 days
$\mathbf{10}^6$	1,6 weeks
$\mathbf{10}^7$	3,8 month
$\mathbf{10}^8$	3,1 years
$\mathbf{10}^9$	31 years
$\mathbf{10}^{10}$	3,1 century
$\mathbf{10}^{11}$	never

Figure 1.8 Seconds table, algorithm speed.

Big O notation is used in computer science to describe the performance or complexity of an algorithm. Actually Big O notation is a special symbol that tells you how fast an algorithm is.

When we define as a rule the worst case assessment and the actual working time should be better. Big O helps to measure algorithm working time when data volume goes bigger. You don't need to count every single operations. Important is only N class (where N is the problem size). When N is small number it is not important what complexity class is an algorithm. Complexi class is important when N is big number. You need to be careful with big complexity algorithms, especially with algorithms $O(2^N)$ classes. When N is big enough you maybe don't see an answer in you lifetime.

We can divide problems into complexity classes:

- O(1) – constant complexity, working time is not dependent on data volume. There is usually a formula to solve.
- O(logN) - log2N complexity, working time grows very slow. Each additional x10 data growing, working time growing only x2.
- O(N) – linear complexity. When data volume grows up two times, working time grows up also two times. For example linear finding.
- O(NlogN) – the same that O(logN), but working time logN grows N times faster.
- $O(N^2)$ – square complexity. As the amount of data increases 10 times the working time increases by $10^2 = 100$ times. In most cases, this algorithm has 2 cycles within each other and both depend on the data. For example several sorting algorithms
- $O(N^3)$ – cube complexity. For example multiplication of matrices.
- $O(2^N)$ – exponential complexity. If N = 10 is time 1000, increasing N by 20 times, the working time increases to 1000000. An impractical algorithm. For example, the solutions of the force method, when all variants must be counted.

computer working speed (operations per seconds)	problem size is 1 million			problem size is 1 billion		
	N	NlgN	N^2	N	NlgN	N^2
10^6	seconds	seconds	**weeks**	hours	hours	**newer**
10^9	now	now	hours	seconds	seconds	**years**
10^{12}	now	now	seconds	now	now	**weeks**

Figure 1.9 Algorithm speed and computing hardness

Algorithm speed and computing hardness (Figure 1.8) show us that fast algorithm with slow computers can solve the problem, but a slow algorithm with a fast computer is useless with a hard problem.

Figure 1.8 shows how analysis of algorithms can help us with solving hard problems more than just a fast computer.

If we find some solutions to make complexity class lower, we can cut algorithm working time from “newer” to “weeks” or if we got some luck to “hours”.

1.3.2 7 Millennium Prize Problems

The Millennium Prize Problems are seven problems in mathematics that were stated by the Clay Mathematics Institute on May 24, 2000[2]. A correct solution to any of the problems results in a 1 million US $ prize being awarded by the institute to the discoverer(s).

1. Poincaré conjecture

Unsolved problems

2. **P-problems versus NP-problems**
3. Hodge conjecture
4. Riemann hypothesis
5. Yang-Mills existence and mass gap
6. Navier–Stokes existence and smoothness
7. Birch and Swinnerton-Dyer Conjecture

As you can see only one of seven Millennium Prize Problems is solved nowadays. This is Poincaré conjecture - solved by Grigori Perelman, Russian scientist (series of articles in 2002-2003).

In the theory of algorithms, the class NP (from the English non-deterministic polynomial) refers to many solvability problems whose solution can be checked on a Turing machine for a time not exceeding the value of a certain polynomial in the size of the input data, if there is some additional information (the so-called solution certificate).

The class P (polynomial) is the set of tasks for which there are “fast” solution algorithms (the operating time of which polynomially depends on the size of the input data). Class P is included in the broader complexity classes of algorithms.

The NP class includes problems that can be solved in polynomial time (class P) on a non-deterministic Turing machine.

Tasks with polynomial-time solution algorithms can be solved using a computer much faster than by direct enumeration, whose time is exponential. This determines the practical significance of the problem of the equality of the classes P and NP.

P-problems versus NP-problems one of the important scientific problems of mathematics. One of the NP-complete problems is the Clique problem. The Millennium Prize Problems list shows to us that the Clique problem is one of the important problem in mathematics and in science at all.

There are main problems that are NP-complete

1. Knapsack problem
2. Graph isomorphism
3. Vertex cover
4. **Clique problem**
5. Hamiltonian path problem
6. Graph coloring
7. Boolean satisfiability (Circuit-SAT, SAT, 3-SAT, etc)

1.3.3 Clique problem

The clique problem belongs to the class of NP-complete problems in graph theory. It was first formulated in 1972 by Richard Karp.

A clique in an undirected graph is a subset of vertices, each two of which are connected by an edge of the graph. In other words, this is a complete subgraph of the original graph. The size of a clique is defined as the number of vertices in it. The clique problem exists in two versions: in the recognition problem, it is necessary to determine whether a clique of size k exists in a given graph G, while in the computational version it is necessary to find a clique of maximum size in a given graph G.

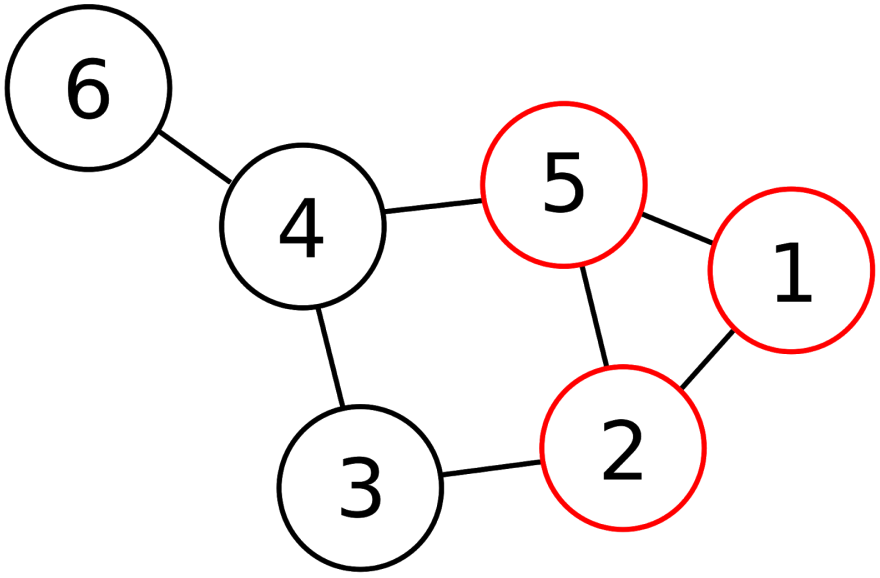


Figure 1.10 Graph with click size 3.

2. Multithreading.

What is multithreading?

In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase the utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores. [4].

Let's make a list of the advantages and disadvantages of multithreading.

Advantages of multithreading:

1. To maintain a responsive user interface
2. To make efficient use of processor time while waiting for I/O operations to complete.
3. To split large, CPU-bound tasks to be processed simultaneously on a machine that has multiple CPUs/cores.

Disadvantages of multithreading:

1. On a single-core/processor machine threading can affect performance negatively as there is overhead involved with context-switching.
2. Have to write more lines of code to accomplish the same task.
3. Multithreaded applications are difficult to write, understand, debug and maintain.

The aim of this work to verify the hypothesis: could maximum clique finding process be accelerated by the multithreading on a single CPU system with processor that have more than 1 core and more than 1 threads.

3. Experiments

3.1 Basic information

This part describes tests and testing results. Also we will make analyses of the results. The results consist of two parts: randomly generated graphs test results figures and DIMACS graph tests result in tables. In the end, we introduce a summary.

In tests we use VRecolor-BT-u, and VRecolor-BT-uVP/VRecolor-BT-uVP2 (code with VRecolor-BT-u algorithm with added parallel calculations) algorithms that was coded in C#.

At the beginning of the work was used ***"starting system",*** but more than almost all tests runs with 100% processor and RAM memory usage. It is ***disadvantage of multithreading*****:** on a single-core/processor machine threading can affect performance negatively as there is overhead involved with context-switching.

Therefore made the decision to upgrade the testing system to the ***"final system"*** with multithreading processor**.**

With a new system (final system), the processor was used on tests not more than 40% and memory usage was not more than 70%. And as result system resources are good enough for provided tests.

The algorithms is coded on C# language, for testing and code modifying used Visual Studio Community 2017 (.Net Framework version 4.5).

Final system:

Processor: Intel Core i3-4130 CPU 3.40GHz (2 cores, 4 threads)

RAM: 12Gb (8Gb+4Gb)

Operation system: 64-bit OS Windows 10 Pro, build 1903

Starting system:

Processor: AMD A6-6400K 3.90GHz (1 cores, 2 threads)

RAM: 8Gb (4Gb+4Gb)

Operation system: 64-bit OS Windows 10 Pro, build 1903

All code is placed in one Microsoft Visual Studio Community 2017 project. For correct working all project files must be in folder **C:\MaxCliqueTest.**

For correct working you need choose settings for Configuration Manager and main files of project, see Figure 3.0

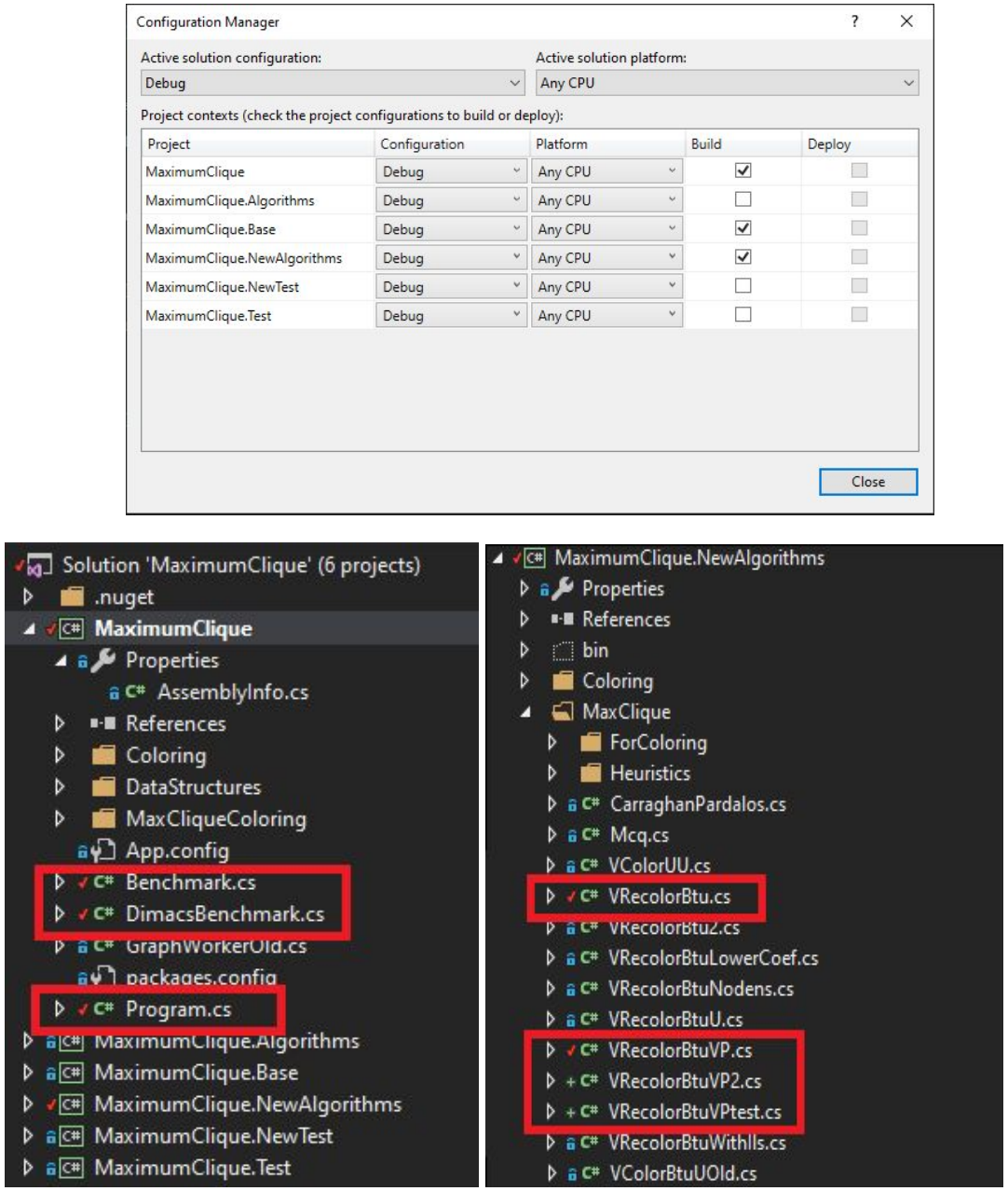


Figure 3.0 Configuration manager settings, and main code files in Visual Studio 2017.

Main files in project that we use in this experiment:

- ***MaximumClique/Program.cs*** - code for main testing environment (we can choose Random or DIMACS graph modes, change options for using test algorithms and make changes for random graphs (number of experiments, graph densities, timeout));
- ***MaximumClique/Benchmark.cs*** - code for starting experiments with random graphs;
- ***MaximumClique/DimacsBenchmark.cs*** - code for starting experiments with DIMACS graphs (folder for DIMACS graphs: C:\MaxCliqueTest\benchmarks\maxClique\tests);
- ***MaximumClique.NewAlgorithms/MaxClique/VRecolorBtu.cs*** - original code for VRecolor-BT-u algorithm.
- ***MaximumClique.NewAlgorithms/MaxClique/VRecolorBtuVP.cs*** - modified code for experiments for this work;
- ***MaximumClique.NewAlgorithms/MaxClique/VRecolorBtuVP2.cs*** - modified code for experiments for this work;
- ***MaximumClique.NewAlgorithms/MaxClique/VRecolorBtuVPtest.cs*** - modified code for experiments for this work

All code is available in GitHub.com (private repositiry), avalable by request: https://github.com/kinoman10/MaxCliqueTest2019/

Basic algorithm VRecolor-BT-u taken from Master's thesis *"Reversed Search Maximum Clique Algorithm Based on Recoloring", Porošin, Aleksandr, TTÜ, 2015 y.* C# code in Appendix 1.

Algorithms were tested on randomly generated graphs. Randomness was generated using Random class from .NET Framework 4.5 which represents a pseudo-random number generator. Figure 3.1 demonstrates a function used for generating random graphs, where Graph is an object containing the adjacency matrix inside the Values array. Generation function takes a number of vertices and density of a graph as parameters and returns a generated graph object.

```
public static Graph GenerateGraph(int nodes, double
density)
{
     int numberOfEdges = Convert.ToInt32(Math.Round(nodes *
            (nodes - 1) * density / 2, 0));
     var graph = new Graph
     {
          Values = new bool[nodes, nodes],
          Edges = numberOfEdges
     };
     var random = new Random();
     Thread.Sleep(40);
     var random2 = new Random();

     int x, y;
     for (int i = 0; i < numberOfEdges; i++)
     {
          do
          {
               x = random.Next(0, nodes);
               y = random2.Next(0, nodes);
          }
          while (x == y || graph.Values[x, y]);
          graph.Values[x, y] = true;
          graph.Values[y, x] = true;
     }
     return graph;
}
```

Figure 3.1 Random graph generation code. (C# language) [A.Porošin]

Also, all marked algorithms were tested on DIMACS graphs. DIMACS = Center for **Di**screte **Ma**thematics and Theoretical **C**omputer **S**cience. This Center presents a pack of different graphs, constructed on the real-life problem basis. List of used DIMACS graphs in Table nr.3.1.

Graph file name	Vertex	Density	Clique size
c-fat500-1.clq	500	0,04	14
c-fat500-2.clq	500	0,07	26
c-fat500-5.clq	500	0,19	64
c-fat500-10.clq	500	0,37	126
hamming6-2.clq	64	0,9	32
hamming6-4.clq	64	0,35	4
hamming8-2.clq	256	0,97	128
hamming8-4.clq	256	0,64	16
hamming10-2.clq	1024	0,99	512
johnson16-2-4.clq	120	0,76	8
johnson8-2-4.clq	28	0,56	4
johnson8-4-4.clq	70	0,77	14
keller4.clq	171	0,65	11
MANN_a27.clq	378	0,99	126
MANN_a9.clq	45	0,93	16
p_hat300-1.clq	300	0,24	8
p_hat300-2.clq	300	0,49	25
p_hat300-3.clq	300	0,74	36
p_hat500-1.clq	500	0,25	9
p_hat500-2.clq	500	0,5	36
p_hat700-1.clq	700	0,25	11
p_hat1000-1.clq	1000	0,25	10
san1000.clq	1000	0,5	15
san200_0.7_1.clq	200	0,7	30
san200_0.7_2.clq	200	0,7	18
san200_0.9_1.clq	200	0,9	70
san200_0.9_2.clq	200	0,9	60
san400_0.5_1.clq	400	0,5	13

Table nr.3.1 - DIMACS graph data, that was using in tests.

Below are explaining how to work VRecolorBtu algorithm, A. Porošin [B.].

CBC – current best clique, largest clique found by so far.

d – depth.

c – index of the currently processed color class.

di – index of the currently processed vertex on depth d.

b – array to save maximum clique values for each color class.

Ca – initial color classes array.

Cb – color classes array recalculated on each depth.

Cd – subgraph of graph G induced by vertices on depth d.

cn – number of color classes recalculated on each depth.

CanBeSkipped(vdi, c) - function that returns true if a vertex can skipped without expanding it.

1. **Graph density calculation.** If graph density is lower than 35% go to step 2a, else go to step 2b.
2. **Heuristic vertex greedy coloring.** There should be two arrays created to store initial color classes defined only once (*Ca*) and color classes recalculated on each depth (*Cb*). During this step, both arrays must be equal.
    a. Before coloring vertices are unordered and colored with swaps.
    b. Before coloring vertices are in decreasing order with response to their degree and colored without swaps.
3. **Searching.** For each color class starting from the first (current color class index *c*).
    3.1. **Subgraph (branch) building.** Build the first depth selecting all the vertices from color classes whose number *c* is equal or smaller than current. Vertices from the first color class should stand first. Vertices at the end should belong to c color class.
    3.2. **Process subgraph.**
    3.2.1**. Initialize depth.** *d = 1*.
    3.2.2. **Initialize current vertex.** Set current vertex index to be expanded (initially the first expanded vertex is the rightmost one). *di + nd* .

3.2.3. **Bounding rule check.** If current branch can possibly contain larger clique than found by so far. If $Ca(v_{di})<c$ and $d-1+b[Ca(v_{di})]<=|CBC|$ then prune. Go to step 3.2.7.

3.2.4. **Vertex skipping check.** If current vertex can possibly contain larger clique than found by so far. If $d-1+Ca(v_{di})$ and *CanBeSkipped( )* skip this vertex. Decrease index $i = i - 1$. Go to step 3.2.3.

3.2.5. **Expand current vertex.** Form new depth by selecting all the adjacent vertices (neighbors) to current vertex $v_{di}(G_{d+1})=N(v_{di})$. Set the next expanding vertex on current depth $di=di-1$.

3.2.6. **New depth analysis.** Check if new depth contains vertices.

a. If $G_{d+1}=\varnothing$ then check if current clique is the largest one it must be saved. Go to step 3.3.

b. If $G_{d+1}=\varnothing$ then check graph density. If graph density is lower than 55% apply greedy coloring with swaps to G_{d+1}, else use greedy coloring without swaps. Save number of color classes *(cn)* acquired by this coloring. If number of color classes cannot possibly give us a larger clique then prune. If $d-1+cn<=|CBC|$ decrease index $i = i - 1$ and go to step 3.2.3, else increase depth $d = d + 1$. Go to step 3.2.2.

3.2.7. **Step back.** Decrease depth $d = d - 1$. Delete expanding vertex from the current depth. If d = 0 go to step 3.3, else go to step 3.2.3.

3.3. **Complete iteration.** Save current best clique value for this color. $b[c] = |CBC|$.

4. **Return maximum clique.** Return CBC.

CanBeSkipped function

th – threshold from which branch will be pruned

CBC – current best clique, largest clique found by so far.

d – depth.

c – index of the currently processed color class.

di – index of the currently processed vertex on depth d.

bnd – bound from which vertices cannot be skipped.

b – array to save maximum clique values for each color class.

Ca – initial color classes array.

Cb – color classes array recalculated on each depth.

1. **Define threshold.** $th=|CBC|-(d-1)$.

2. **Find skipping bound.** For each vertex index *dj* from *di – 1* to 0. If $Ca(v_{dj})<c$ and $b[Ca(v_{dj})<=th]$ then $bnd = j$.

3. **Decide whether vertex can be skipped**. For each adjacent (to currently expanded) vertex with index *dj* from *bnd* to zero. If $Ca(v_{dj})>th$ then return false. If $Cb(v_{dj})>th$ had never occurred return true.

3.2 Adding parallel computing in code (experiment 1)

In this part we describe where we put parallel methods in basic algorithm. We have basic parts in code: main algorithm code, coloring and recoloring parts.

protected override void Solution() - main part of code, for all graphs

private void InitialColoring() - for graph with density < 0.35
private void InitialColoringWithSwaps() - for graph with density > 0.35

private int Recolor(int depth) - for graph with density < 0.55
private int RecolorWithSwaps(int depth) - for graph with density > 0.55

Density of graph $D\frac{2|E|}{|V|(|V|-1)}$, where D - density of graph, E-edge numbers of graph, V-number of vertices in the graph.

As a first step we find places where we can use standard C# Parallel method from namespace: System.Threading.Tasks for `InitialColoring()`:

```
private void InitialColoring()
...
for (int i = 0; i < NodesNumber; i++)
{...}
for (int i = 0; i < NodesNumber; i++)
{
      for (int j = i + 1; j < NodesNumber; j++)
      {...}
}
…
for (int i = 0; i < NodesNumber; i++)
{     for (int j = 0; j < initialColorsNumber; j++)
      {     for (int k = 0; k < initialNodesNumInColorClass[j];
            k++)
            {...break;}
      ...
      }
}
```

Figure 3.2 Private void InitialColoring() (pseudo code)

```
private void InitialColoring()
...
Parallel.For (int i = 0; i < NodesNumber; i++)
{...}
Parallel.For (int i = 0; i < NodesNumber; i++)
{
      for (int j = i + 1; j < NodesNumber; j++)
      {...}
}
...
for (int i = 0; i < NodesNumber; i++)
{     for (int j = 0; j < initialColorsNumber; j++)
      {     for (int k = 0; k < initialNodesNumInColorClass[j];
            k++)
            {...break;}
      ...
      }
}
```

Figure 3.3 Private void InitialColoring() with added C# Parallel method (pseudo code).

As a second step we places where we can use standard C# Parallel method from namespace: System.Threading.Tasks for `InitialColoringWithSwaps()`:

```
private void InitialColoringWithSwaps()
...
for (...)
{...
}
while()
{...
      for (...)
      {
            for (...)
            {
            …
            break;
            }...
      }
      …
}
```

Figure 3.4 Private void InitialColoringWithSwaps() (pseudo code).

```
private void InitialColoringWithSwaps()
…
Parallel.For (...)
{
      ...
}
while()
{
      ...
      for (...)
      {
            for (...)
            {
            ...
            break; // break for while
            }
      ...
      }
      ...
}
```

Figure 3.5 Private void InitialColoringWithSwaps() with added C# Parallel method (pseudo code).

As a third step we find places where we can use standard C# Parallel method from namespace: System.Threading.Tasks for `RecolorWithSwaps()`:

```
private int RecolorWithSwaps(int depth)
...
for (int i = 0; i < length; i++)
{
…
}
while (true)
{
      ...
      for (int i = colored; i < length; i++)
      {
            ...
            for (int j = lowerBound; j < colored; j++)
```

```
            {
                  if (Graph.Values[array[i] - 1, array[j] - 1])
                  {...
                  break;  // break for while
                  }
            }
      if (colored == length)
      {
            ...
            break;  // break for while
      }
      ...
}
```

Figure 3.6 Private void RecolorWithSwaps() (pseudo code).

```
private int RecolorWithSwaps(int depth)
...
Parallel.For(int i = 0; i < length; i++)
{
…
}
while (true)
{
      ...
      for (int i = colored; i < length; i++)
      {
            ...
            for (int j = lowerBound; j < colored; j++)
            {
                  if (Graph.Values[array[i] - 1, array[j] - 1])
                  {...break;  // break for while}
            }
      if (colored == length)
      {
            ...
            break;  // break for while
      }
      ...
}
```

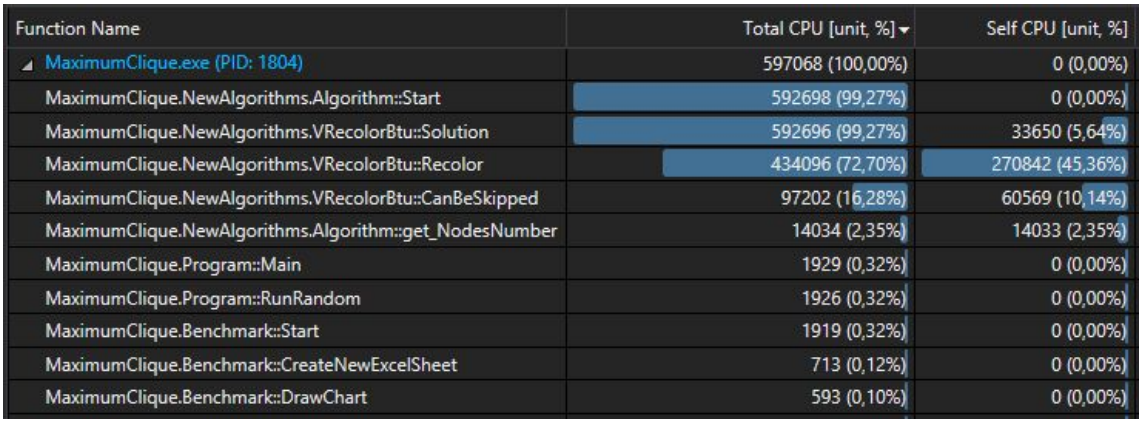
Figure 3.7 Private void RecolorWithSwaps() with added C# Parallel method (pseudo code).

Other part of code doesn't give to use C# Parallel method without fundamental changes in code of VRecolor-BT-u algorithm.

All figure with algorithm woking (calculating) times you can see in "4.1 Random generated graphs test results for Adding parallel computing in code

3.3 Parallel methods in code based on CPU usage profile (experiment 2).

In this part we try to find functions that use more CPU resource than other functions and put some Parallel methods for using threads and core of our testing system.



Function Name	Total CPU [unit, %]	Self CPU [unit, %]
MaximumClique.exe (PID: 1804)	597068 (100,00%)	0 (0,00%)
MaximumClique.NewAlgorithms.Algorithm::Start	592698 (99,27%)	0 (0,00%)
MaximumClique.NewAlgorithms.VRecolorBtu::Solution	592696 (99,27%)	33650 (5,64%)
MaximumClique.NewAlgorithms.VRecolorBtu::Recolor	434096 (72,70%)	270842 (45,36%)
MaximumClique.NewAlgorithms.VRecolorBtu::CanBeSkipped	97202 (16,28%)	60569 (10,14%)
MaximumClique.NewAlgorithms.Algorithm::get_NodesNumber	14034 (2,35%)	14033 (2,35%)
MaximumClique.Program::Main	1929 (0,32%)	0 (0,00%)
MaximumClique.Program::RunRandom	1926 (0,32%)	0 (0,00%)
MaximumClique.Benchmark::Start	1919 (0,32%)	0 (0,00%)
MaximumClique.Benchmark::CreateNewExcelSheet	713 (0,12%)	0 (0,00%)
MaximumClique.Benchmark::DrawChart	593 (0,10%)	0 (0,00%)

Figure 3.8 Profile for VRecolorBtu algorithm, dencity 90%

Solution() - main part of code, for all graphs, **Recolor()** - for graph with density < 0.55, **RecolorWithSwaps()** - for graph with density > 0.55, **CanBeSkipped()** - code for all graphs.

We make 25 sets for each density, for getting real using CPU data.

Random graph dencity, %	Total CPU [unit, %]			
	Solution()	RecolorWithSwaps()	Recolor()	CanBeSkipped()
10%	43,07%	17,55%		4,94%
20%	62,74%	23,15%		20,20%
30%	69,15%	33,21%		13,70%
40%	72,68%	39,43%		12,38%
50%	73,97%	40,62%		14,99%
60%	77,32%		51,07%	13,92%
70%	80,21%		54,87%	15,40%
80%	99,39%		67,69%	18,82%
90%	99,27%		72,70%	16,28%

Table 3.2 Total CPU [unit, %] for original VRecolorBtu code, random graphs.

Random graph dencity, %	Self CPU [unit, %]			
	Solution()	RecolorWithSwaps()	Recolor()	CanBeSkipped()
10%	16,98%	16,19%		4,61%
20%	16,75%	21,70%		18,80%
30%	19,53%	31,43%		12,81%
40%	18,41%	37,70%		11,66%
50%	16,30%	38,74%		14,12%
60%	10,02%		33,97%	13,92%
70%	8,80%		36,71%	14,18%
80%	7,65%		40,73%	12,38%
90%	5,64%		45,36%	10,14%

Table 3.3 Self CPU [unit, %] for original VRecolorBtu code, random graphs.

All DIMACS graphs Table nr.3.1	Self CPU [unit, %]			
	Solution()	RecolorWithSwaps()	Recolor()	CanBeSkipped()
Total CPU, %	94,94%	7,48%	78,92%	2,46%
Self CPU, %	16,75%	7,44%	51,89	2,46%

Table 3.4 Self CPU [unit, %] for original VRecolorBtu code, DIMACS graphs.

From tables 3.1-3.4 we can see that Recolor() and RecolorWithSwaps() procedures takes more CPU resources that others.

In profiler we find that more than 20% resources takes 3-4 strings of code.

```
for (int k = 0; k < nodesNumInColorClass[j]; k++)
     {
     if (Graph.Values[vert - 1, colorClasses[j][k] - 1])
          {
               connected = true;
               break;
          }
```

Figure 3.9 Code that takes almost 15% of CPU usage from VRecolorBtu, Recolor().

```
var resRecolor = Parallel.For(0, nodesNumInColorClass[j], new
ParallelOptions { MaxDegreeOfParallelism = 4}, (k, loopStateK)
=>
{
     if (Graph.Values[vert - 1, colorClasses[j][k] - 1])
     {
                loopStateK.Break();
           }
});
if (!resRecolor.IsCompleted){ connected = true;}
```

Figure 3.10 New code with Parallel method in VRecolorBtuVPtest, Recolor().

```
bool canBeColored = true;
for (int j = lowerBound; j < colored; j++)
if (Graph.Values[array[i] - 1, array[j] - 1])
   {
     canBeColored = false;
      break;
   }
```

Figure 3.11 Code that takes almost 15% of CPU usage VRecolorBtu, RecolorWithSwaps().

```
bool canBeColored = true;
var resW = Parallel.For(lowerBound, colored, new
ParallelOptions { MaxDegreeOfParallelism = 2 }, ( j,
loopStateJ) =>
{
     if (Graph.Values[array[i] - 1, array[j] - 1])
     {
          loopStateJ.Break();
     }
});
if (!resW.IsCompleted)
{canBeColored = false;}
```

Figure 3.12 New code with Parallel method in VRecolorBtuVPtest, RecolorWithSwaps().

3.4 Adding threads elements in code (experiment 3, VRecolBtuVPtest code)

After code analysis we found that main time algorithm use in part of code in void Solution() in **while circle**. Pseudo code of Solution() is bellow.

```
protected override void Solution()
{...
for (int c = 0; c < initialColorsNumber; c++)
{     for (int i = 0; i <= c; i++)
      {...
            for (int j = 0; j < initialNodesNumInColorClass[i];
                  j++)
            {...}
      }
      while (depth >= 0)
            {
            ...
            if (inDepthIndex == -1)
                  {...
                  continue;// continue for while
                  }
            ...
            if (color < c + 1 ... maxCliqueSize)
                  {...
                  continue;//continue for while
                  }
            if ((depth + ... c + 1))
                  {...
                  continue;//continue for while
                  }
            ...
            for (int i = 0; i < inDepthIndex; i++)
            {     if (Graph... - 1])
                        {...
                        }
            }

            for (int i = ... i--)
                 {if (Graph...- 1])
                        {...
                        }
                 }
            ...
            if (numberOfNodesArr[depth] > 0)
                  {...
```

```
                if (depth + colNum <= maxCliqueSize)
                      {
                     ...
                     }
               else
                      {
                      if (depth > maxCliqueSize)
                            {...
                          break;//break for while
                          }
                              depth--;
                 }
                 ...
           }
          cache[c] = maxCliqueSize;
     }
}
```

Figure 3.13 Protected override void Solution() (pseudo code).

As the beginning we made code refactoring for easer use Threads methods for parts of code. And run part of "void Solution()" as threads.

```
protected override void Solution()
{
…
     for (int c = 0; c < initialColorsNumber; c++)
     {
     SolutionForFor(c, depth);
     inDepthElementIndex[depth] = numberOfNodesArr[depth] - 1;
     depth = SolutionWhile(c, depth);
     }
     cache[c] = maxCliqueSize;

}
```

Figure 3.14 Protected override void Solution() after code refactoring (pseudo code).

```
protected override void Solution()
{
...
     for (int c = 0; c < initialColorsNumber; c++)
     {
     Thread[] threads = new Thread[initialColorsNumber];
     Thread[] threadss = new Thread[initialColorsNumber];
     threads[c] = new Thread(NILL =>
     {
     SolutionForFor(c, depth);
```

```
        });

        threads[c].Priority = ThreadPriority.Highest;
        threads[c].Start();
        threads[c].Join();

        inDepthElementIndex[depth] = numberOfNodesArr[depth] - 1;

        threadss[c] = new Thread(NILL =>
        {

        depth = SolutionWhile(c, depth);

        });
        threadss[c].Priority = ThreadPriority.Highest;
        threadss[c].Start();
        threadss[c].Join();

        }
        cache[c] = maxCliqueSize;
}
```

Figure 3.15 Protected override void Solution() after code refactoring using Threads (pseudo code).

All figure with algorithm woking times you can see in “4.2 Random generated graphs test results (code with Threads).

4 Results

4.1 Random generated graphs test results (experiment 1)

For testing random generated graphs w take 2 algorithms VRecolor-BT-u (by A. Porošin 2015) and VRecolor-BT-u with added some code for parallel counting (VRecolor-BT-u-VP). All code modifications described in Experiments “3.1 Adding parallel computing in code ”.

In this test blocks we were made 26 test sessions with random generated graphs. Graphs have density from 10% to 90%. C# code for generating random graphs you can see on Figure 4.1 - Figure 4.9

If we find during tests that ower new code gives some good dynamics on some densities, we will make additional tests with this tensitys and will use graphs with larger number of vertices.

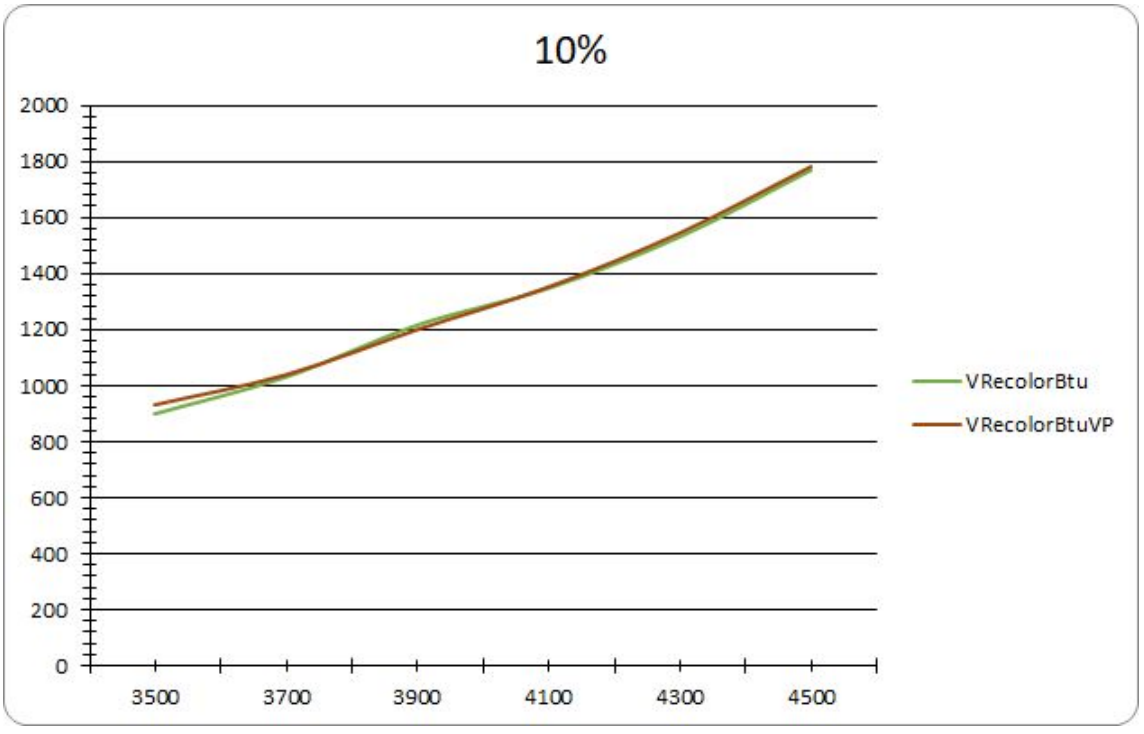


Figure 4.1 Randomly generated graphs test (Parallel). Density 10%.

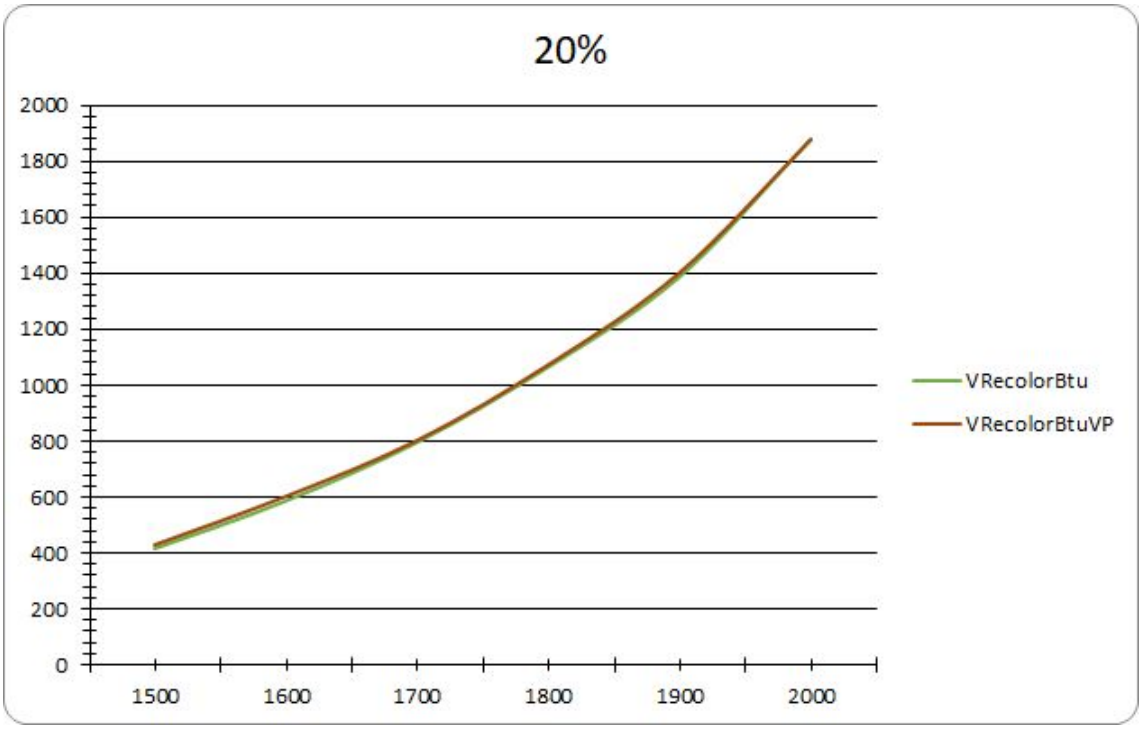


Figure 4.2 Randomly generated graphs test (Parallel). Density 20%.

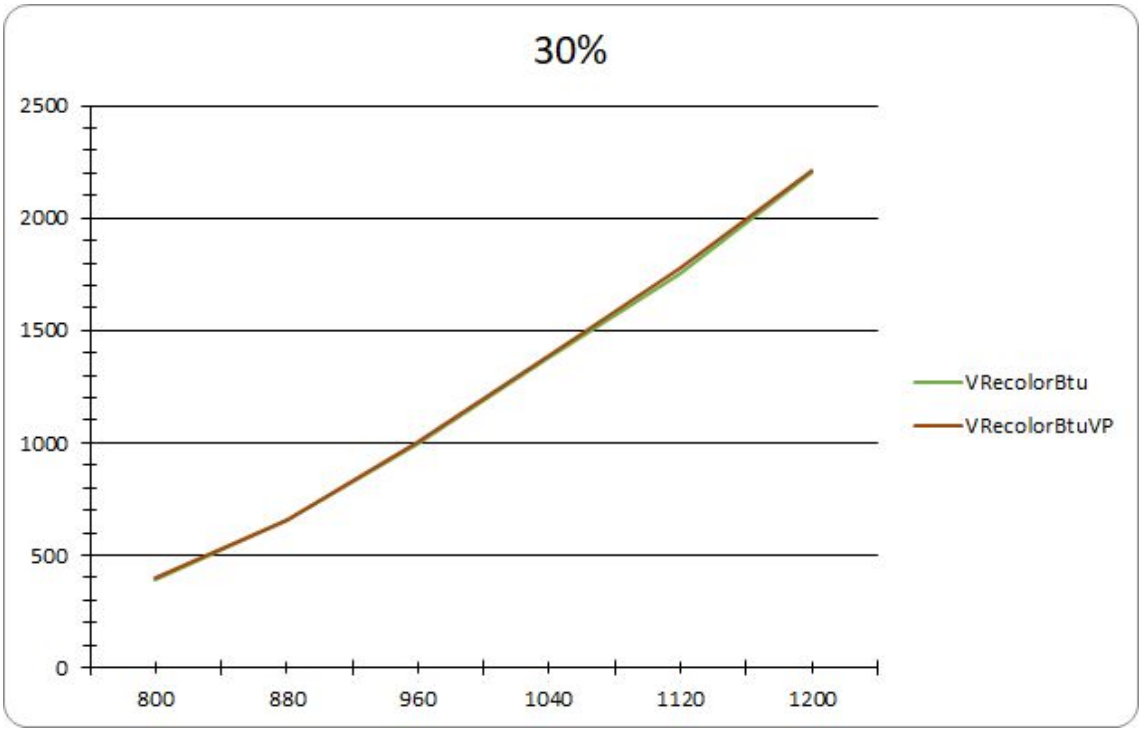


Figure 4.3 Randomly generated graphs test (Parallel). Density 30%.

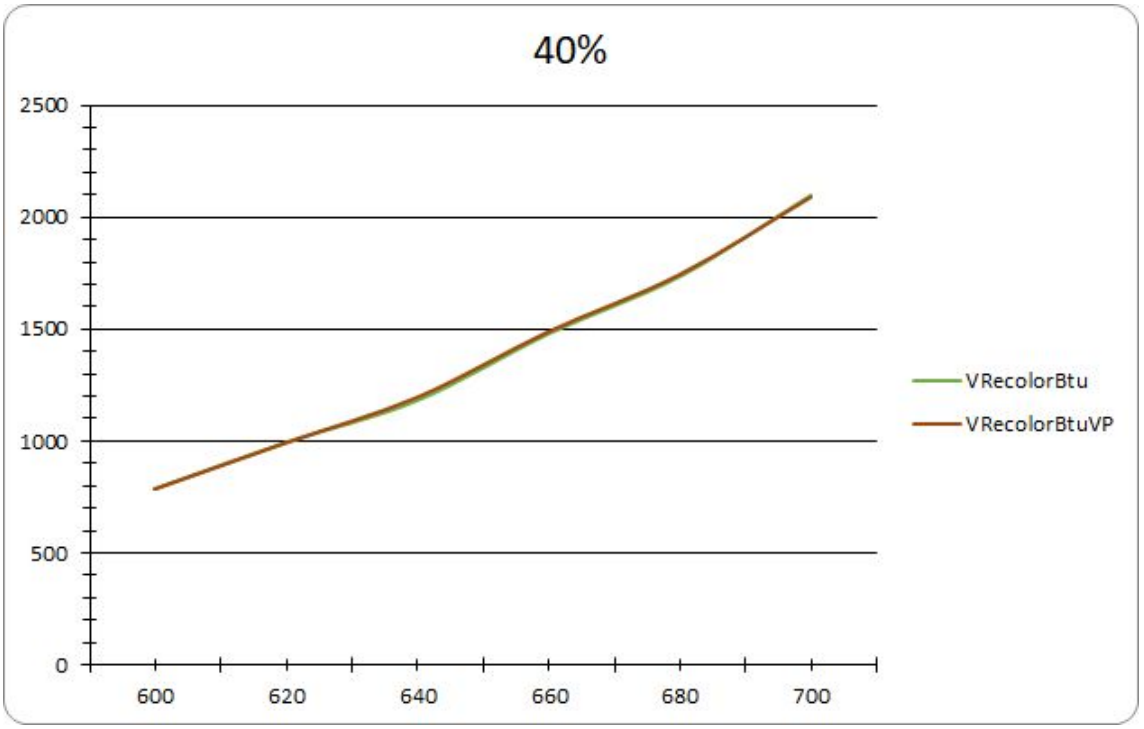


Figure 4.4 Randomly generated graphs test (Parallel). Density 40%.

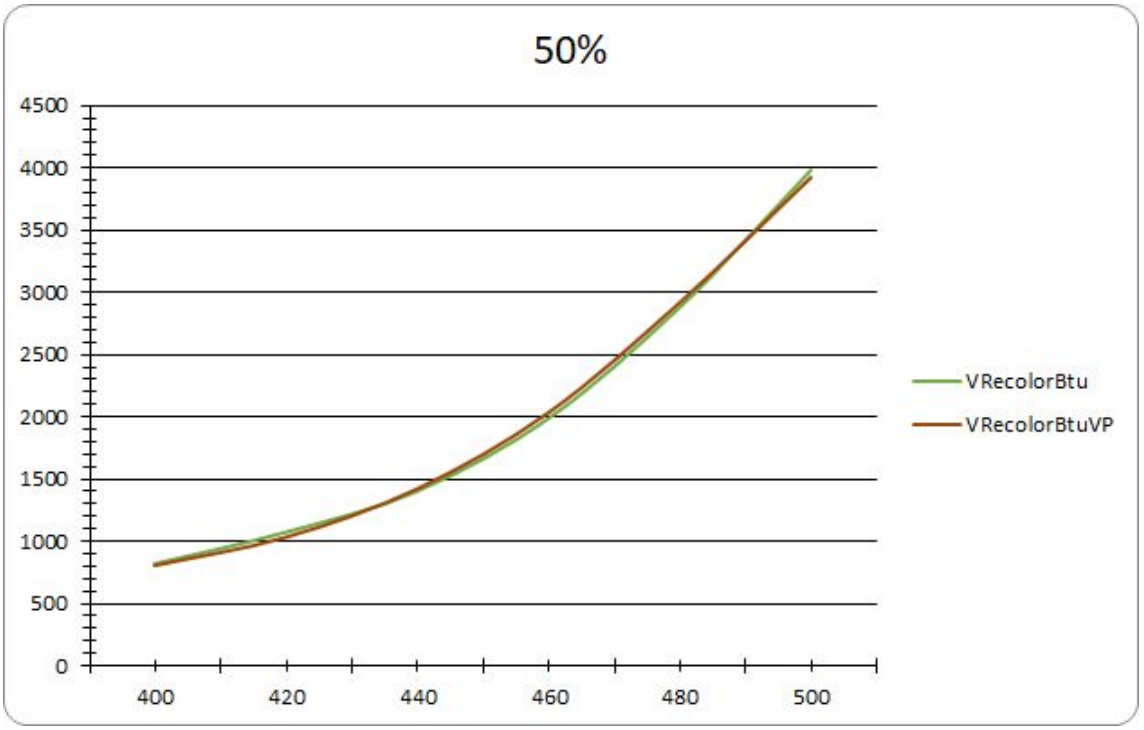


Figure 4.5 Randomly generated graphs test (Parallel). Density 50%.

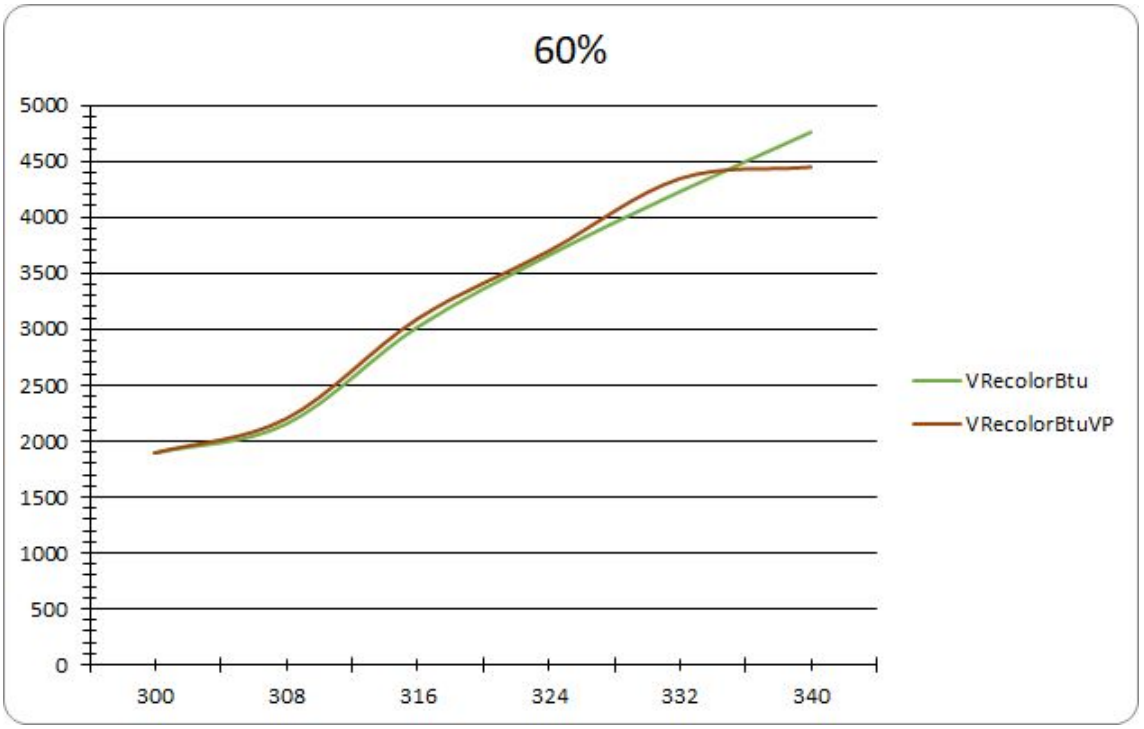


Figure 4.6 Randomly generated graphs test (Parallel). Density 60%.

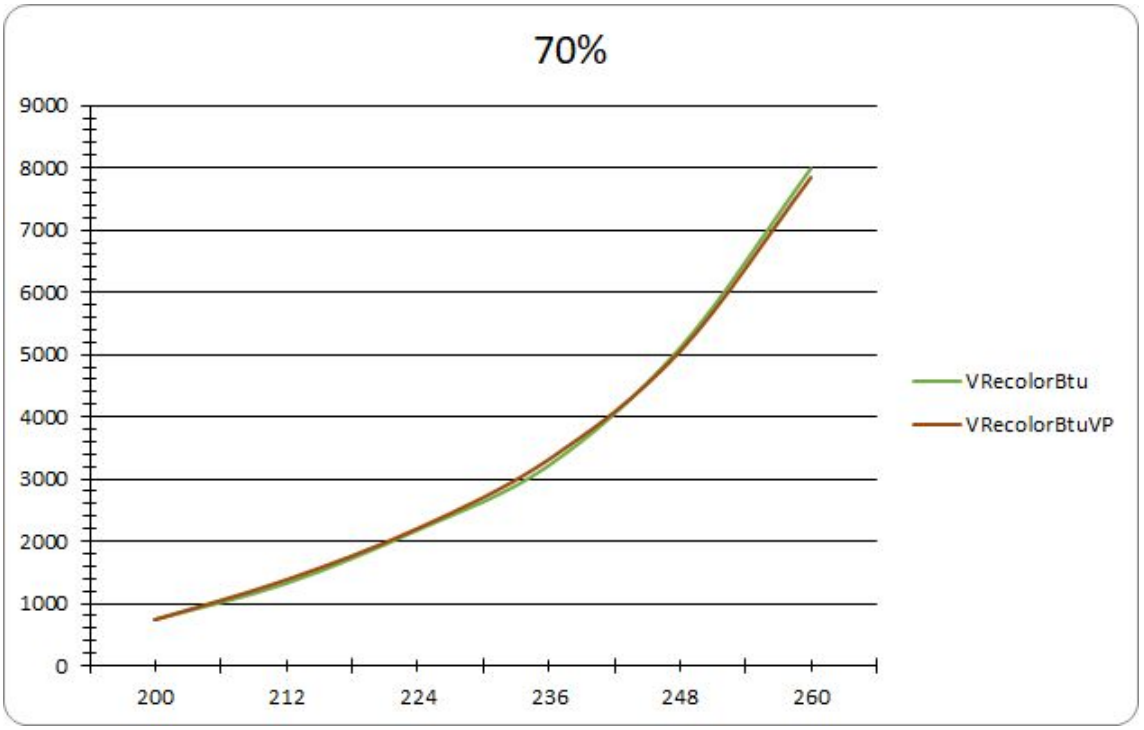


Figure 4.7 Randomly generated graphs test (Parallel). Density 70%.

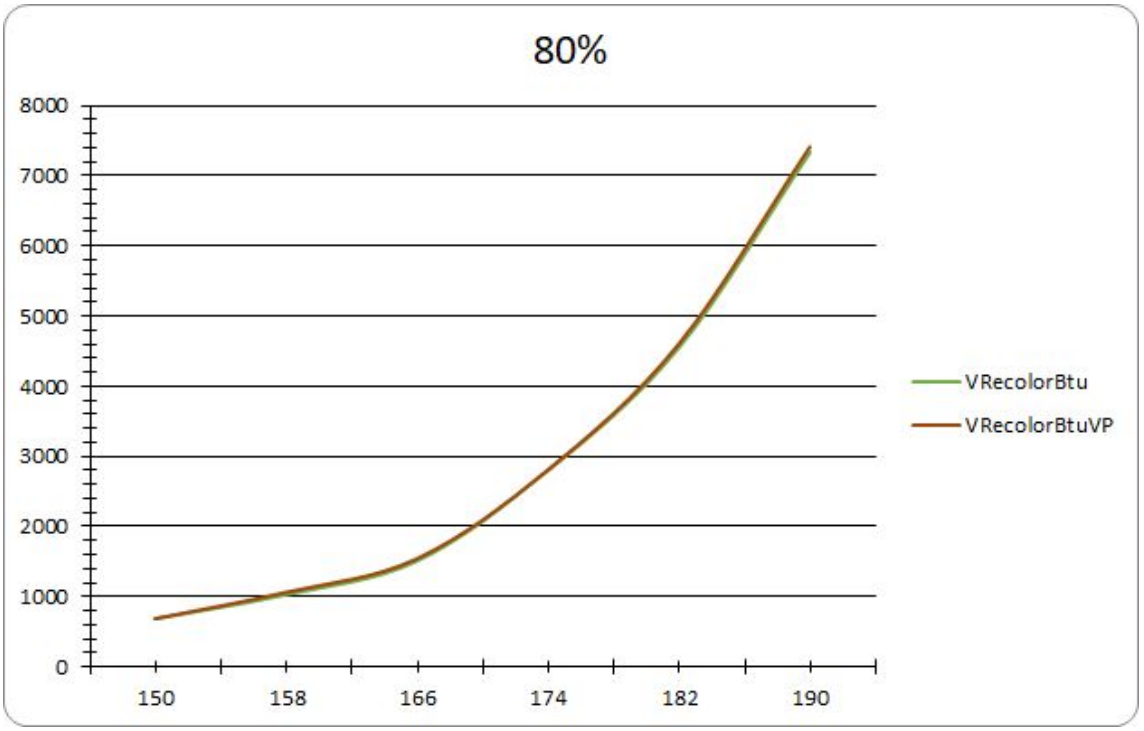


Figure 4.8 Randomly generated graphs test (Parallel). Density 80%.

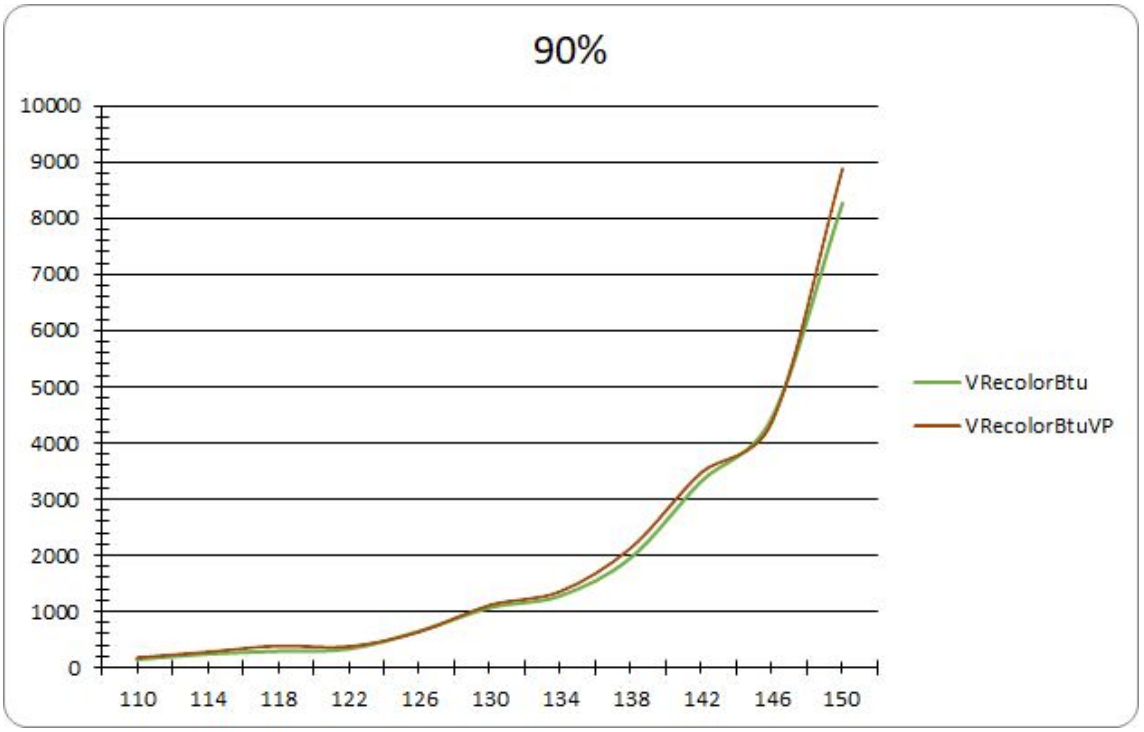


Figure 4.9 Randomly generated graphs test (Parallel). Density 90%.

As result parallel counting with using Parallel.For methods doesn't give any good enough results. On random graphs parallel counting gives some minimum advantage on graph with 70-90% densities.

Almost on all Figures 4.1-4.9 algoritms VColor-BT-u and VColor-BT-u-VP gives the same results.

4.2 Random generated graphs test results (code with Threads, experiment 2).

At the beginning we need to mark that our Final system with i3 gives not result at all. All time processor was on 100% usage and results were beyond reasonable time.

All figures below i can get with testing system:

Processor: Intel Core i7-8500K CPU 4.30GHz (6 cores, 12 threads)

RAM: 32Gb (16Gb+16Gb)

Operation system: 64-bit OS Windows 10 Pro

Unfortunately we can do only random graph tests on this powerful system, for DIMACS tests this system was unavailable for reasons beyond our control.

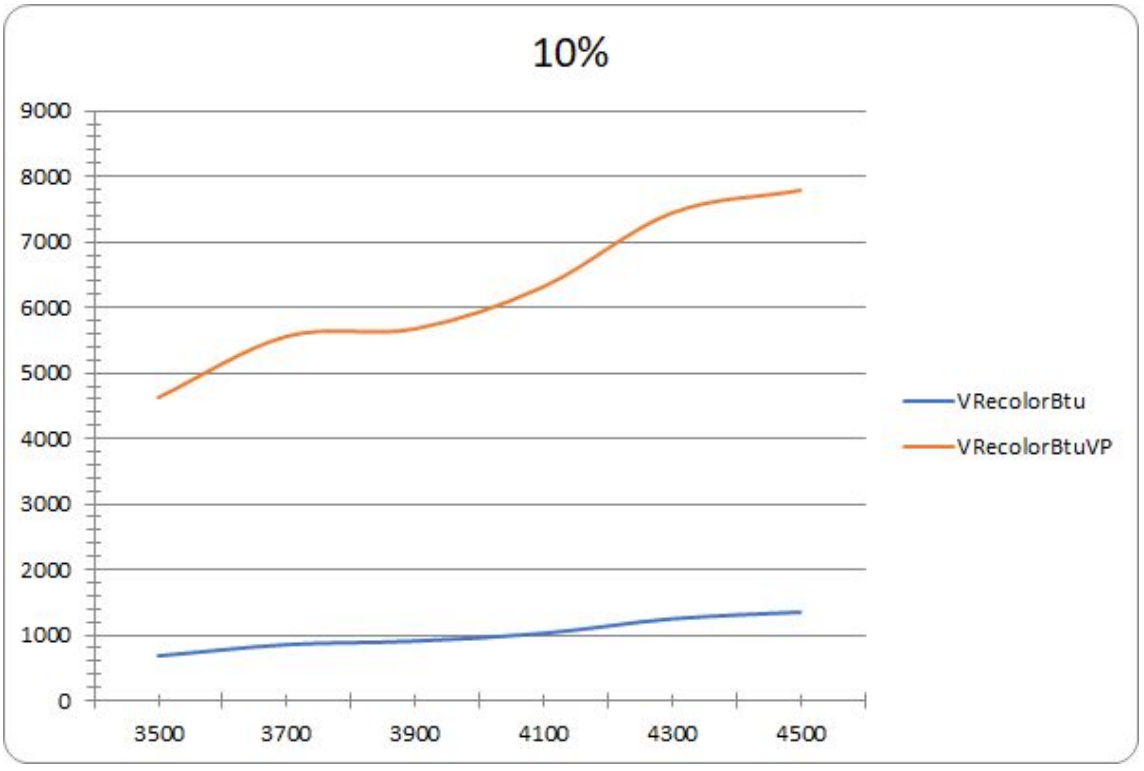


Figure 4.10 Randomly generated graphs test (Parallel+Threads). Density 10%.

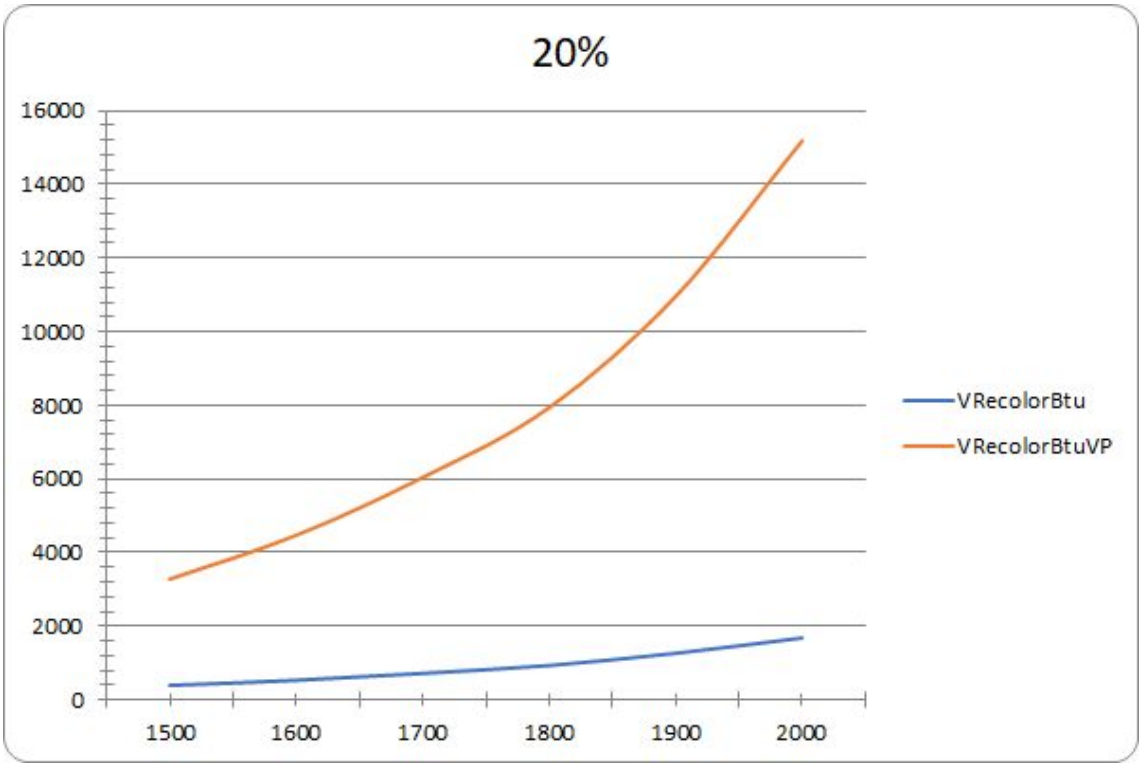


Figure 4.11 Randomly generated graphs test (Parallel+Threads). Density 20%.

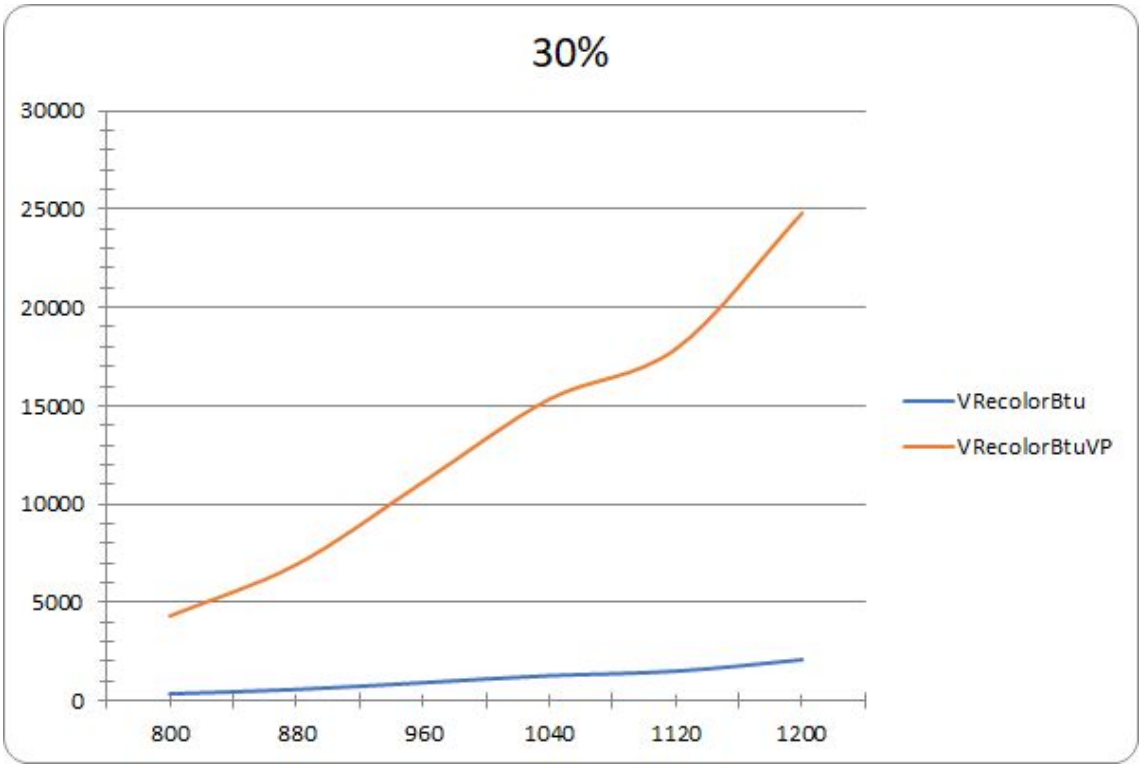


Figure 4.12 Randomly generated graphs test (Parallel+Threads). Density 30%.

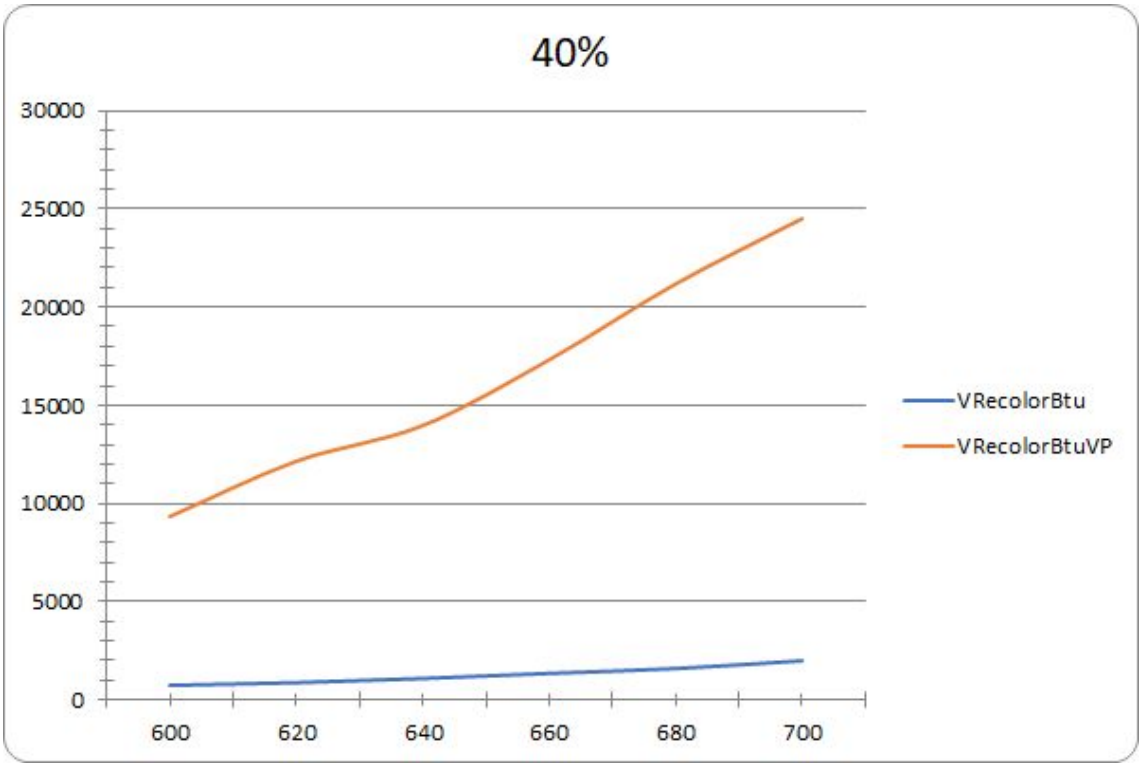


Figure 4.13 Randomly generated graphs test (Parallel+Threads). Density 40%.

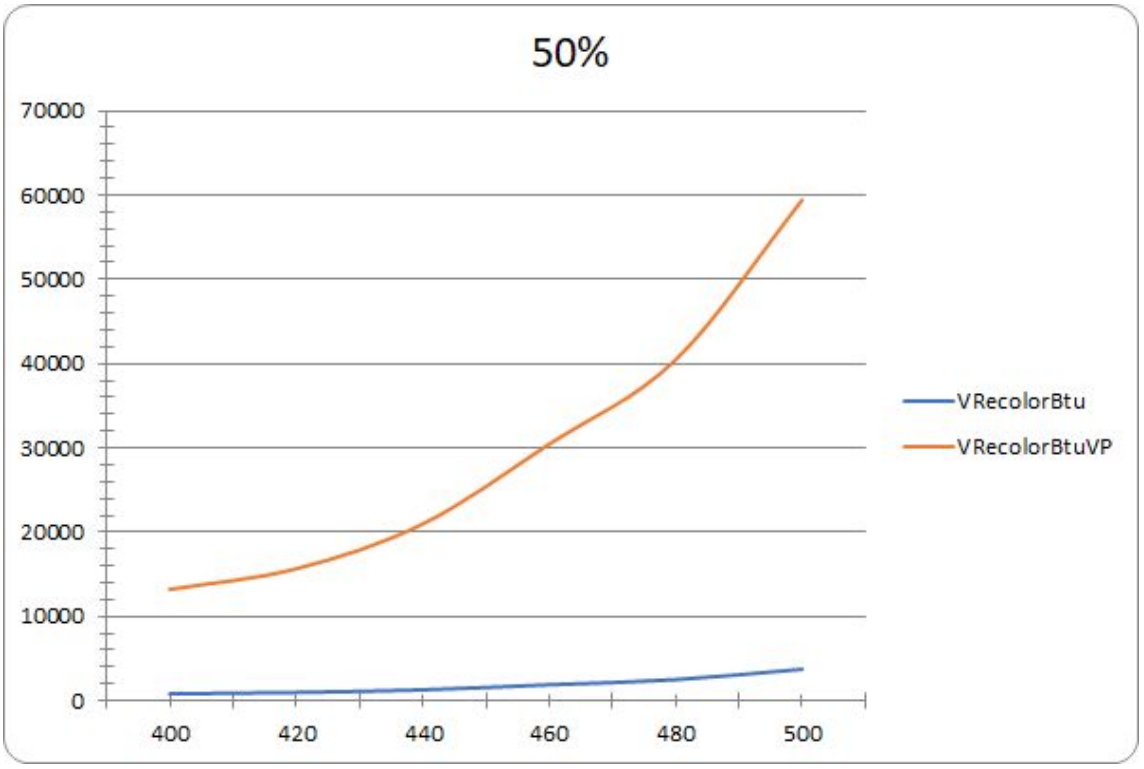


Figure 4.14 Randomly generated graphs test (Parallel+Threads). Density 50%.

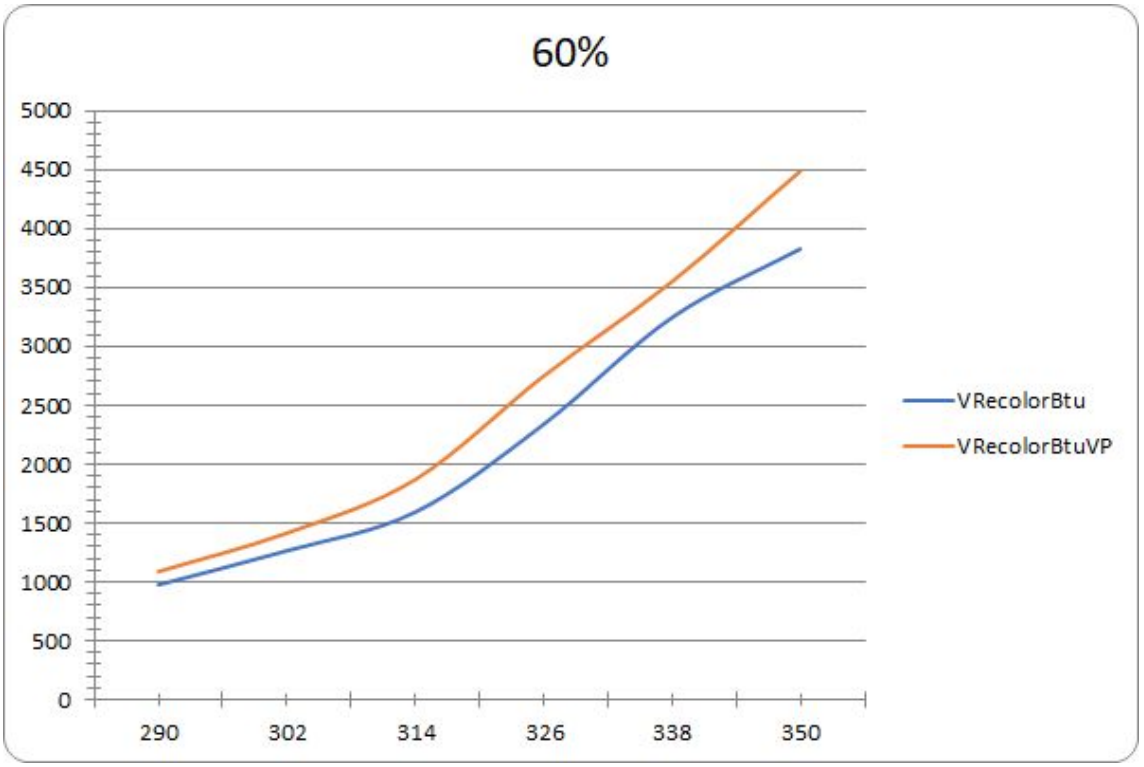


Figure 4.15 Randomly generated graphs test (Parallel+Threads). Density 60%

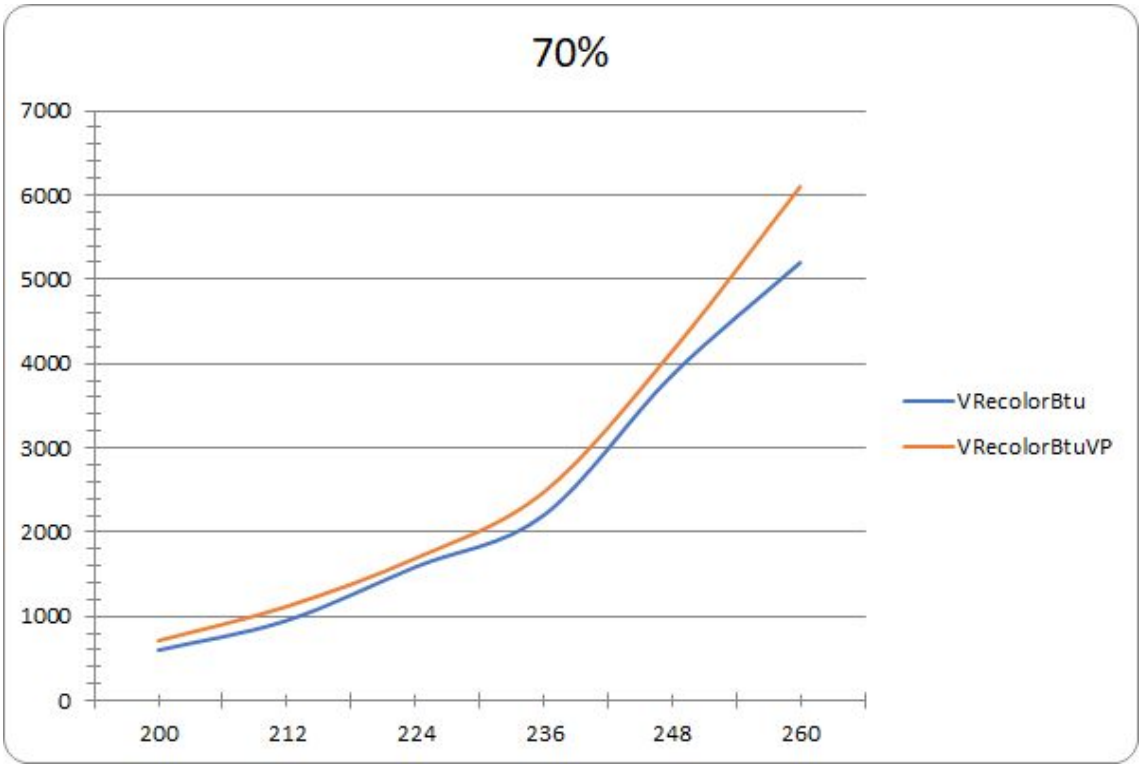


Figure 4.16 Randomly generated graphs test (Parallel+Threads). Density 70%

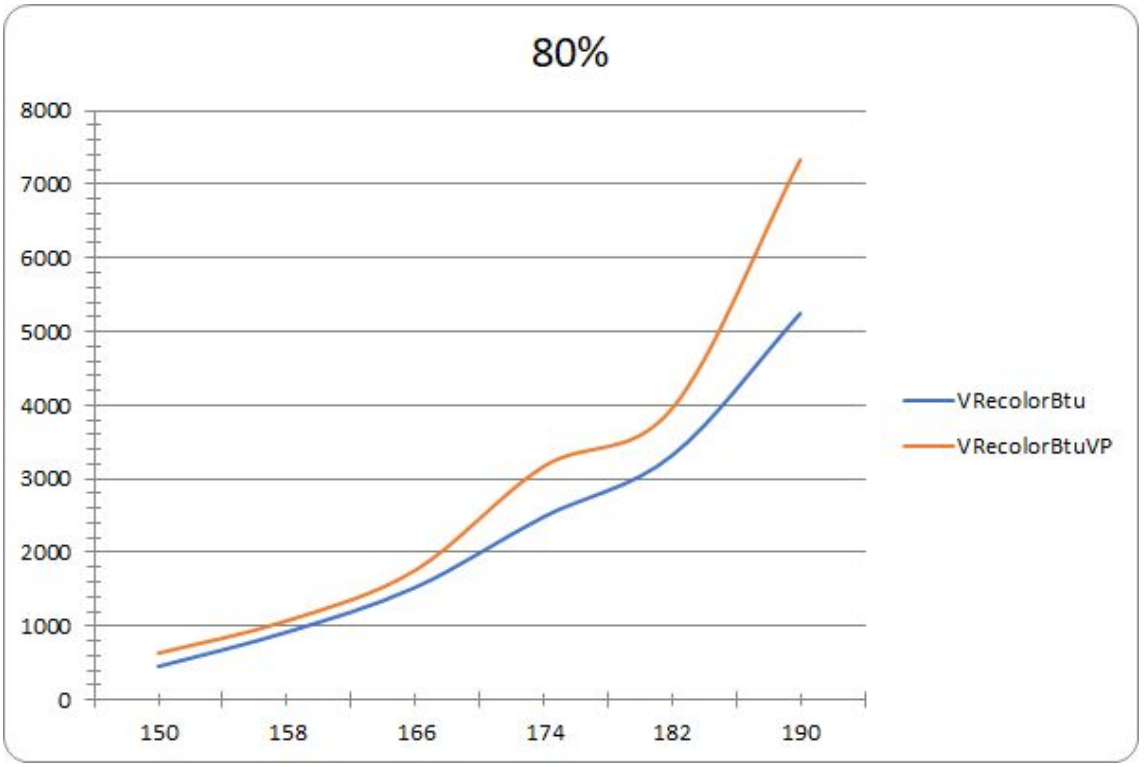


Figure 4.17 Randomly generated graphs test (Parallel+Threads). Density 80%

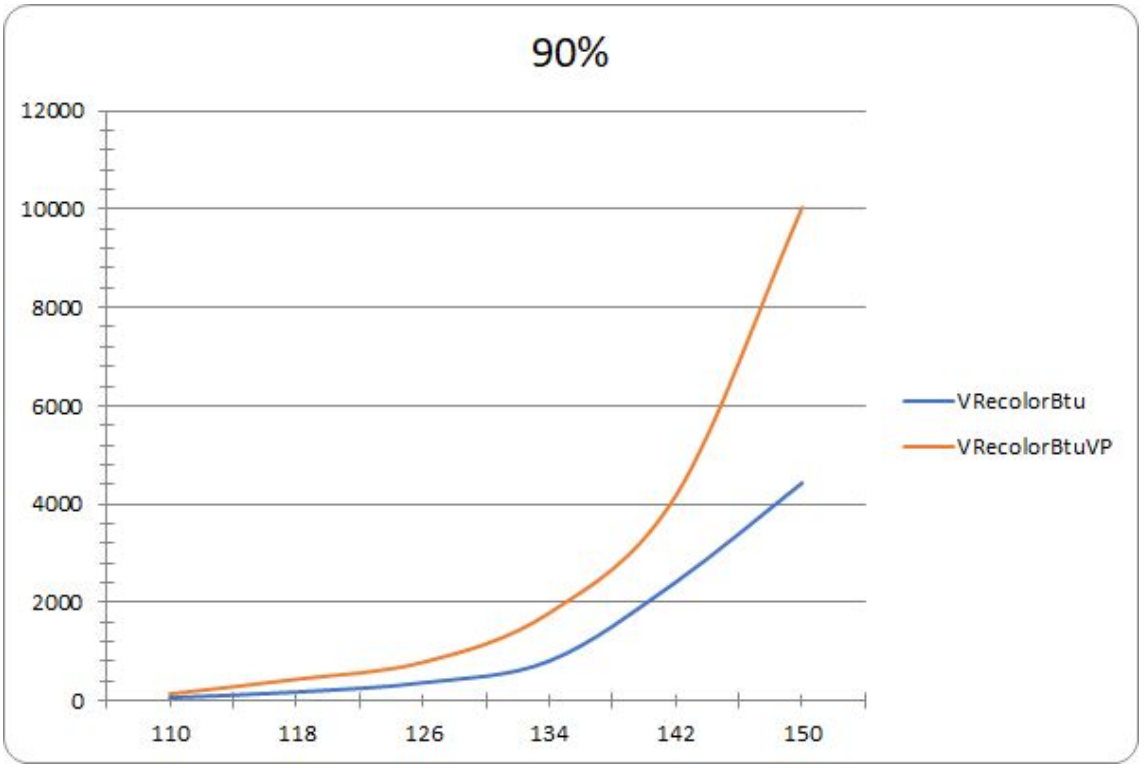


Figure 4.18 Randomly generated graphs test (Parallel+Threads). Density 90%

As you can see on Figure 4.19 when PC working Threads in code, processor usage was 100%, but on density 60-80% time line on figure moved very close to base algorithm time. And if we have a little bit more CPU power maybe we can get positive result for that test (best time result for calculating).

I think that in this case we need some smart strategy for using threads, but now i have not enough experience in multi thread programming in C#.

All code that i was fund is sample examples for Thread using in C# code. Some complicated codes samples are not in public usage. More tries is not so good as 2-3 years of programming experience in real tasks. The algorithm optimization problem is trivial problem.

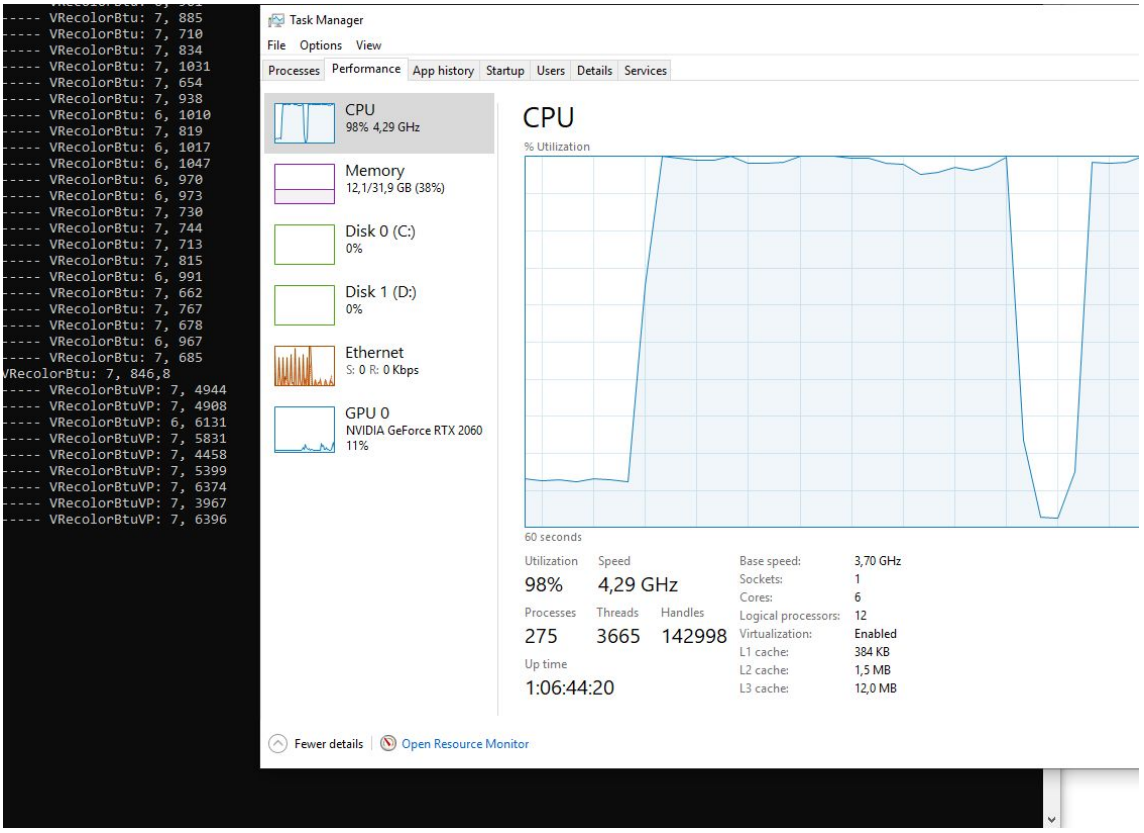


Figure 4.19 Processor usage on random graph, code with threads using.

4.3 Random generated graphs test results (Parallel methods in code based on CPU usage profile, experiment 3).

After profiling on random graphs we make changes in code that must help to make calculation time smaller.

All reasonable variants was tested:

- was taken graphs with density from 10 to 90%;
- was tested graphs with small vertices number;
- were tested DIMACS graphs.

As a result random graphs take all 100% CPU power and do not give any result.

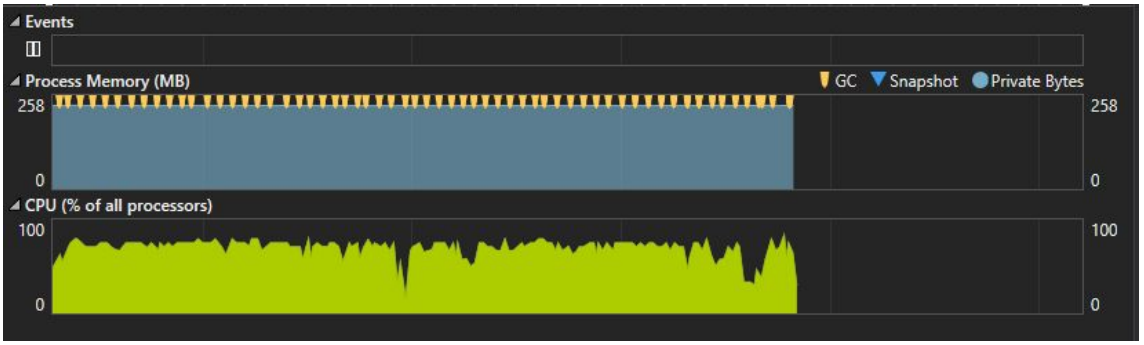


Figure 4.20 100% CPU utilization on testing random graphs with VRecolorBTUtest.

For some reason 15 DIMACS graphs gives result, and you can see it on

4.4 DIMACS graphs test results (for all algorithm codes).

In this subchapter, three algorithms were tested on DIMACS graphs.There are VRecolor-BT-u (by A. Porošin 2015) and VRecolor-BT-uVP with added some code for parallel counting. All code modifications described in Experiments “3.1 Adding parallel computing in code”.

				VRecolor-Btu	VRecolor-BtuVP	VRecolor-BtuVP2	VRecolor-BtuVPtest
Graph	Nodes	Density	Max Clique	Time	Time	Time	Time
c-fat500-1	500	0,04	14	**4**	72	219	195
c-fat500-2	500	0,07	26	**2**	26	974	413
c-fat500-5	500	0,19	64	**13**	55	645	**n/a**
c-fat500-10	500	0,37	126	**114**	424	1629	**n/a**
hamming6-2	64	0,90	32	**2**	4	316	1212
hamming6-4	64	0,35	4	**0**	1	78	48
hamming8-2	256	0,97	128	418	**239**	1405	**n/a**
hamming8-4	256	0,64	16	**12**	208	277	4247
hamming10-2	1024	0,99	512	**62039**	64544	70452	**n/a**
johnson8-2-4	28	0,56	4	**0**	8	80	365
johnson8-4-4	70	0,77	14	**1**	2	174	505
johnson16-2-4	120	0,76	8	**548**	654	660	**n/a**
keller4	171	0,65	11	**71**	82	657	12360
MANN_a9	45	0,93	16	1	**0**	198	256
MANN_a27	378	0,99	126	12028	**11014**	12532	**n/a**
p_hat300-1	300	0,24	8	**12**	138	374	2212
p_hat300-2	300	0,49	25	**233**	924	619	69818
p_hat300-3	300	0,74	36	15800	**15082**	17025	**n/a**
p_hat500-1	500	0,25	9	**118**	**442**	426	8726
p_hat500-2	500	0,50	36	**7290**	22006	7977	**n/a**
p_hat700-1	700	0,25	11	**225**	1054	688	23192
p_hat1000-1	1000	0,24	10	**1977**	8395	2496	103722
san1000	1000	0,50	15	**807**	1647	1227	**n/a**
san200_0.7_1	200	0,70	30	1651	**76**	2151	**n/a**
san200_0.7_2	200	0,70	18	10	**7**	151	**n/a**
san200_0.9_1	200	0,90	70	55	**44**	661	**n/a**
san200_0.9_2	200	0,90	60	1466	**1352**	2005	**n/a**
san400_0.5_1	400	0,50	13	**30**	58	156	5933

Table 4.1 DIMACS graph results. Time consumption (ms).

Table 4.1 shows results of DIMACS graph testing, with red color marked graphs where algorithm wins.

On DIMACS real-life problem graphs we have wins on some graph not more than 5-7%, on graphs with 70-90% density.

In this case parallel counting takes more CPU resources that gives advantages, therefore almost all tests with VRecolor-BT-u-VP take more time that original VRecolor-BT-u code. But VRecolor-BT-u-VP with easiest parallel methods gives 7 wins.

VRecolor-BT-u-VP2 and VRecolor-BT-u-VPtest takes no wins.

Original VRecolor-BT-u algorithm is good enough optimized for real life DIMACS graphs without using multi thread counting, and we need to find other ways for making code or algorithm faster. There is no easy way.

5. Conclusions

5.1 Summary

In C# language we have built-in tools (modules) for working with threads. Each thread starting, working and managing by Operating System (in our case Windows 10 Pro 64-bit). Therefore not so important how many threads and cores we have physical in CPU. More importing how we can use and manage threads.

Operating System initialise threads by itself for working of many applications in one time and regulate virtual threads for physical CPU threads and cores (and how it really works is black box for us).

Speed of thread work depends on thread priority level in system (from 0 to 31 in Windows OS). If we don't have enough resources, threads with the lowest (and even highest) priority wait in queue for free system resources. OS threads always have higher priority than users program (threads).

Multithreading gives advantages in working time for simple operations circles. Multithreading is especially good for acceleration on linear algorithms such as sorting and simple searching. For more complicated algorithms, not all operations can be performed in parallel or/and multithreaded mode.

In sorting operation we have **counting number of elements**. When we take graph we have **counting number of elements**, **counting number of connections**, but we need to find **unknown number of complete subgraph with unknown size**. That unknowns complicates max clique algorithm and makes it nonlinear (nonlinear dependence of the number of graph vertices and maximum click size).

In regular algorithm optimization gives more result that increasing CPU power.. Basic (clique) algorithm was modified at least 2 times (were added coloring and recoloring procedure). Modifications gives way to minimize calculation for finding maximum clique and makes working time for algorithm lower.

Simple adding Parallel methods in calculation give not positive result in this experiment, because our algorithm Recolor-BT-u is good enough optimized for working on PC (Windows OS) from the beginning. And if we want to use Parallel and Threads

method with this algorithm we need fundamental rewrite all algorithm code and optimise it for multithread technology in C# language.

Adding Thread method taking a lot of PC resources for working and even powerful PC (that i can find and use one time) with i7 (6 cores, 12 threads)+32gb RAM doesn't give positive results. For most experiments it takes 100% of resources so we can't exactly say does theads give some positive results for time or not. But our simple method with using Parallel methods gives some result with DIMACS graphs.

Some process in code can not be calculated in parallel procedures (like Parallel.For or Parallel.ForEach), because process don't have fixed count of circles. Circles have conditions, that modified in current circle and the number of cycles depends on it.

Parallel.For, Parallel.ForEach methods is simple and powerful methods, that gives possibility to regulate Maximum Degree Of Parallelism.

We can lock some variables to make parallel calculation without risk.

In our case, the processing of additional threads requires more time than we get as a gain from multithreaded calculations. The more vertices we have on the graph, the more time is wasted using multithreading.

5.2 Future studies

Multithreaded programming - a new promising way to use power of new generation multicore processors. Multithreaded programming is not so simple, that we can use it without studies and practicing.

As future studies we have opportunities for optimise the Recolor-BT-u algorithm, (variable and objects, and other data structures) using capabilities and limitations of multithreading/parallel programming for C # and OS Windows 10 pro (64-bit).

Another way with big potential: using the power of multi-core graphics card architecture (for example CUDA of Nvidia for Visual Studio). Graphic cards can use they own memory and free main PC memory for other processes.

CUDA from nVidia gives possibility to use the power of several graphic cards with multi-core processors (up to some thouthands) with a separate (fast) memory.

Model	NVIDIA GTX 1080 Ti
Multiprocessors	64
CUDA cores	3584
Производительность, TFLOPS	11,34
Memory (GDDR5X), Gb	11

Table 5.1 Preferences of Graphic Card NVIDIA GTX 1080Ti.

CUDA (Compute Unified Device Architecture) is a hardware-software architecture for parallel computing, which can significantly increase computing performance through the use of Nvidia GPUs.

5.3 Kokkuvõte.

C # keeles on olemas sisseehitatud tööriistad (moodulid) thread-i töötamiseks. Iga thread-i käivitamine, töötamine ja haldamine opsüsteemi abil (meie puhul 64-bitine Windows 10 Pro). Seetõttu pole nii oluline, kui palju lõimeid ja südamikke meil protsessoris füüsiliselt on. Rohkem importi, kuidas saame lõime kasutada ja hallata.

Operatsioonisüsteem initsialiseerib lõimed iseenesest paljude rakenduste korraga töötamiseks ja reguleerib füüsilisi protsessori lõimede ja südamike virtuaalseid niite (ja kuidas see tegelikult töötab, on meie jaoks must kast).

Niitide töö kiirus sõltub lõime prioriteetsuse tasemest süsteemis (0 kuni 31 Windows OS-is). Kui meil pole piisavalt ressursse, oodake madalaima (ja isegi kõrgeima) prioriteediga niidid tasuta süsteemiressursside saamiseks järjekorras. OS-i niidid on alati kõrgema prioriteediga kui kasutajaprogrammid (threads).

Mitmekeermelisus annab lihtsa toiminguga ringidele tööaja eelised. Multithreading on eriti hea kiirenduseks lineaarsete algoritmide korral, näiteks sortimine ja lihtne otsimine. Keerukamate algoritmide puhul ei saa kõiki toiminguid teostada paralleelses või / ja mitmekihilises režiimis.

Sorteerimise käigus on meil loendatud arv elemente. Kui võtame graafiku, on meil loendatud elementide arv, ühenduste arv, kuid peame leidma teadmata arvu tundmatu suurusega alamgraafi. See tundmatu muudab keerukamaks maksimaalse klikkide algoritmi ja muudab selle mittelineaarses (graafi tippude arvu ja maksimaalse klõpsu suuruse mittelineaarne sõltuvus).

Tavalise algoritmi optimeerimine annab rohkem tulemusi kui protsessori võimsuse suurendamine. Põhialgoritmi (clique) muudeti vähemalt 2 korda (lisati värvimis- ja värvimisprotseduur). Modifikatsioonid annavad võimaluse minimeerida arvutamist **maximum clique** leidmiseks ja vähendavad algoritmi tööaega.

Paralleelsete meetodite lihtne lisamine arvutamisel ei anna selles katses positiivset tulemust, kuna meie algoritm Recolor-BT-u on algusest peale piisavalt optimeeritud töötama arvutis (Windows OS). Ja kui me tahame selle algoritmiga kasutada paralleel- ja lõimemeetodit, peame kogu algoritmi koodi põhjalikult ümber kirjutama ja optimeerima selle mitmeharuliseks tehnoloogiaks C # keeles.

Multithread-i lisamine, mis võtab palju arvutiressursse töötamiseks ja isegi võimsa arvuti jaoks (mida ma suudan korraga leida ja kasutada) koos i7-ga (6 core, 12 thread) + 32 GB RAM-i, ei anna positiivseid tulemusi. Enamiku katsete jaoks kulub 100% ressurssidest, nii et me ei saa täpselt öelda, kas multithread-i meetodid annavad positiivseid tulemusi või mitte. Kuid meie lihtne meetod paralleelsete meetodite kasutamisel annab DIMACS-i graafikutega teatud tulemuse.

Mõningaid koodiprotsesse ei saa arvutada paralleelsete protseduuridega (näiteks Parallel.For või Parallel.ForEach), kuna protsessil pole fikseeritud ringide arvu.

Parallel.For, Parallel.ForEach meetodid on lihtsad ja võimsad meetodid, mis annavad võimaluse reguleerida paralleelsuse, lisaks saame lukustada mõned muutujad, et teha paralleelne arvutus ilma riskita.

Meie puhul nõuab täiendavate thread-i töötlemine rohkem aega, kui me võtame multithread-i arvutuste abil. Mida rohkem tippe graafikul on, seda rohkem aega kasutatakse multithread-ile.

6. References

A. D. Kumlander, Some Practical Algorithms to Solve The Maximum Clique Problem, Tallinn, 2005.

B. “Pöördotsingu suurima kliki leidmise algoritm ülevärvimise teel. Reversed Search Maximum Clique Algorithm Based on Recoloring”, Porošin, Aleksandr, TTÜ, Magistritöö, 2015 a.

1. Alexanderson, Gerald (July 2006). ”Euler and Königsberg's bridges: a historical view”. *Bulletin of the American Mathematical Society*. 43 (4): 567.
2. Arthur M. Jaffe "The Millennium Grand Challenge in Mathematics", "Notices of the AMS", June/July 2000, Vol. 53, Nr. 6, p. 652-660
3. Robert Sedgewick, Kevin Wayne “Algorithms (4th Edition)”, Addison-Wesley Professional; 4th edition (March 19, 2011) Language: English ISBN: 978-0-321-60633-4, (Russian Edition, ISBN: 978-5-8459-2070-6 pp. 44-59)
4. Multithreading (computer architecture), https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)
5. Graph Theory — History & Overview, https://towardsdatascience.com/graph-theory-history-overview-f89a3efc0478
6. Richard Branson - https://twitter.com/richardbranson, 04:18 - 3.sentember.2012.

F1. MAA Euler Archive

F2. https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

F3. Two equivalent ways to draw graph, https://www.researchgate.net/figure/Two-equivalent-ways-to-draw-our-orthogonality-graph-Since-orthogonal_fig1_234526050

F4. P, NP, NP-complete NP-hard.svg, https://en.wikipedia.org/wiki/File:P_np_np-complete_np-hard.svg

Appendix 1

Base class Algorithm, by A. Porošin 2015:

```
using System.Diagnostics;
using MaximumClique.Base;
using MaximumClique.NewAlgorithms.Coloring;
using System.Threading;

namespace MaximumClique.NewAlgorithms
{
    public abstract class Algorithm
    {
        protected readonly Stopwatch _stopwatch = new
     Stopwatch();
        private bool _solved;
        protected Graph Graph;

        public ColoringAlgorithm Coloring { get; set; }

        protected Algorithm(Graph graph)
        {
            Graph = graph;
            SolutionFoundElapsed = 1;
        }

        #region Properties

        public bool IsSolved
        {
            get
            {
                return _solved;
            }
        }

        public double Elapsed
        {
            get { return _stopwatch.ElapsedMilliseconds; }
        }

        protected long branches;
        public long Branches
        {
            get { return branches; }
        }

        public double SolutionFoundElapsed { get; protected
     set; }
```

```
        private int _nodesNumber = -1;
        protected int NodesNumber
        {
            get
            {
                return _nodesNumber == -1 ? (_nodesNumber =
     Graph.Values.GetLength(0)) : _nodesNumber;
            }
        }

        public abstract int Result { get; }

        #endregion //Properties

        protected abstract void Solution();

        public void Start()
        {
            _stopwatch.Start();

                Solution();

            _stopwatch.Stop();
            _solved = true;
        }
    }
}
```

VRecolor-BT-u algorithm, by A. Porošin 2015:

```
using System.Linq;
using MaximumClique.Base;

namespace MaximumClique.NewAlgorithms
{
    public class VRecolorBtu : Algorithm
    {
        private int maxCliqueSize;
        private int[,] levelNodes;
        private int initialColorsNumber;
        private int[][] initialColorClasses;
        private int[] initialNodesNumInColorClass;
        private int[] initialColors;
        private int[,] inDepthColors;
        private int[] numberOfNodesArr;
        private int[] inDepthElementIndex;
        private int[,] skippedNodes;
        private int[] skippedNodesNumber;
```

```
    private int[] cache;

    public VRecolorBtu(Graph graph)
        : base(graph)
    {
        levelNodes = new int[NodesNumber, NodesNumber];
        initialColorClasses = new int[NodesNumber][];
        initialNodesNumInColorClass = new int[NodesNumber];
        initialColors = new int[NodesNumber];
        inDepthColors = new int[NodesNumber, NodesNumber];
        numberOfNodesArr = new int[NodesNumber + 1];
        inDepthElementIndex = new int[NodesNumber + 1];

        skippedNodes = new int[NodesNumber, NodesNumber];
        skippedNodesNumber = new int[NodesNumber + 1];
        cache = new int[NodesNumber];
    }

    public override int Result
    {
        get { return maxCliqueSize; }
    }

    protected override void Solution()
    {
        if (Graph.Density < 0.35)
            InitialColoringWithSwaps();
        else
            InitialColoring();

        var inDepthDegree = new int[NodesNumber];
        for (int c = 0; c < initialColorsNumber; c++)
        {
            skippedNodesNumber[0] = 0;
            int depth = 0;
            numberOfNodesArr[depth] = 0;
            inDepthDegree[depth] = 0;
            for (int i = 0; i <= c; i++)
            {
                for (int j = 0; j <
initialNodesNumInColorClass[i]; j++)
                {
                    levelNodes[depth,
numberOfNodesArr[depth]] = initialColorClasses[i][j];
                    numberOfNodesArr[depth]++;
                }
            }
            inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;
            while (depth >= 0)
            {
```

```
                int inDepthIndex =
inDepthElementIndex[depth];
                if (inDepthIndex == -1)
                {
                    depth--;
                    continue;
                }
                int p = levelNodes[depth, inDepthIndex];
                var color = initialColors[p - 1];
                if (color < c + 1 && depth + cache[color -
1] <= maxCliqueSize)
                {
                    depth--;
                    continue;
                }
                if ((depth + inDepthColors[depth, p - 1]
                    <= maxCliqueSize) &&
CanBeSkipped(inDepthIndex, depth, c + 1))
                {
                    skippedNodes[depth,
skippedNodesNumber[depth]] = p;
                    skippedNodesNumber[depth]++;
                    inDepthElementIndex[depth]--;
                    continue;
                }
                branches++;
                int prevDepth = depth;
                depth++;
                numberOfNodesArr[depth] = 0;
                inDepthElementIndex[depth] = 0;

                for (int i = 0; i < inDepthIndex; i++)
                {
                    if (Graph.Values[levelNodes[prevDepth,
inDepthIndex] - 1, levelNodes[prevDepth, i] - 1])
                    {
                        levelNodes[depth,
numberOfNodesArr[depth]] = levelNodes[prevDepth, i];
                        numberOfNodesArr[depth]++;
                    }
                }
                for (int i = skippedNodesNumber[prevDepth]
- 1; i >= 0; i--)
                {
                    if (Graph.Values[levelNodes[prevDepth,
inDepthIndex] - 1, skippedNodes[prevDepth, i] - 1])
                    {
                        levelNodes[depth,
numberOfNodesArr[depth]] = skippedNodes[prevDepth, i];
                        numberOfNodesArr[depth]++;
                    }
```

```
                }

                inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;

                if (numberOfNodesArr[depth] > 0)
                {
                    int colNum = Graph.Density < 0.55 ?
RecolorWithSwaps(depth) : Recolor(depth);

                    if (depth + colNum <= maxCliqueSize)
                        depth--;
                }
                else
                {
                    if (depth > maxCliqueSize)
                    {
                        maxCliqueSize = depth;
                        break;
                    }
                    depth--;
                }
                inDepthElementIndex[prevDepth]--;

            }
            cache[c] = maxCliqueSize;
        }
    }

    private bool CanBeSkipped(int vertIndex, int depth, int
currentColor)
    {
        int threshold = maxCliqueSize - depth;
        int vert = levelNodes[depth, vertIndex];

        // on current depth on what vertex index we will
cut??
        // if (color < c + 1 && depth + cache[color - 1] <=
maxCliqueSize){ depth--; }
        int initialColorThresholdIndex = -1;
        for (int i = vertIndex - 1; i >= 0; i--)
        {
            int vert2 = levelNodes[depth, i];
            if (initialColorThresholdIndex == -1)
            {
                int color = initialColors[vert2 - 1];
                if (color < currentColor && cache[color -
1] <= threshold)
                {
                    initialColorThresholdIndex = i;
                    break;
```

```
                }
            }
        }

        // if we want to skip a vertex, we have to check if
current vertex
        // is adjacent to any vertex INSIDE vertices that
will be CUT!! (from initialColorThresholdIndex to zero)
on this depth
        // with color higher than threshold
        for (int i = initialColorThresholdIndex; i >= 0;
i--)
        {
            int vert2 = levelNodes[depth, i];
            if (Graph.Values[vert - 1, vert2 - 1])
            {
                if (inDepthColors[depth, vert2 - 1] >
threshold)
                    return false;
            }
        }
        return true;
    }

    public int FindNumberOfColorClasses(int depth, int
numberOfNodes)
    {
        int nPrevColor = 0;

        int numberOfColorClasses = 0;
        for (int i = 0; i < numberOfNodes; i++)
        {
            int currentColor =
initialColors[levelNodes[depth, i] - 1];
            if (currentColor != nPrevColor)
            {
                numberOfColorClasses++; nPrevColor =
currentColor;
            }
        }
        return numberOfColorClasses;
    }

    private void InitialColoring()
    {
        var verticesWithDegrees = new int[NodesNumber][];
        // count degrees of vertices

        for (int i = 0; i < NodesNumber; i++)
        {
            verticesWithDegrees[i] = new int[2];
```

```
            verticesWithDegrees[i][0] = i + 1;
        }

        for (int i = 0; i < NodesNumber; i++)
            for (int j = i + 1; j < NodesNumber; j++)
                if (Graph.Values[i, j])
                {
                    verticesWithDegrees[i][1]++;
                    verticesWithDegrees[j][1]++;
                }

        // order vertices by degree
        var orderedVertices =
verticesWithDegrees.OrderByDescending(i =>
i[1]).ToArray();

        // color vertices, find color classes
        for (int i = 0; i < NodesNumber; i++)
        {
            int vert = orderedVertices[i][0];
            bool isAdded = false;
            for (int j = 0; j < initialColorsNumber; j++)
            {
                bool connected = false;
                for (int k = 0; k <
initialNodesNumInColorClass[j]; k++)
                {
                    if (Graph.Values[vert - 1,
initialColorClasses[j][k] - 1])
                    {
                        connected = true;
                        break;
                    }
                }
                if (!connected)
                {

initialColorClasses[j][initialNodesNumInColorClass[j]] =
vert;
                    initialColors[vert - 1] = j + 1;
                    inDepthColors[0, vert - 1] = j + 1;
                    initialNodesNumInColorClass[j]++;
                    isAdded = true;
                    break;
                }
            }
            if (!isAdded)
            {
                initialColorClasses[initialColorsNumber] =
new int[NodesNumber];
```

```
initialColorClasses[initialColorsNumber][initialNodesNumI
nColorClass[initialColorsNumber]] = vert;

initialNodesNumInColorClass[initialColorsNumber]++;
                initialColorsNumber++;
                initialColors[vert - 1] =
initialColorsNumber;
                inDepthColors[0, vert - 1] =
initialColorsNumber;
            }
        }
    }

    private void InitialColoringWithSwaps()
    {
        var array = new int[NodesNumber];
        for (int i = 0; i < NodesNumber; i++)
            array[i] = i + 1;

        int colored = 0;
        initialColorsNumber = 0;

        while (true)
        {
            initialColorClasses[initialColorsNumber] = new
int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumI
nColorClass[initialColorsNumber]] = array[colored];

initialNodesNumInColorClass[initialColorsNumber]++;
            initialColorsNumber++;
            initialColors[array[colored] - 1] =
initialColorsNumber;
            inDepthColors[0, array[colored] - 1] =
initialColorsNumber;

            colored++;
            int lowerBound = colored - 1;
            for (int i = colored; i < NodesNumber; i++)
            {
                bool canBeColored = true;
                for (int j = lowerBound; j < colored; j++)
                    if (Graph.Values[array[i] - 1, array[j]
- 1])
                    {
                        canBeColored = false;
                        break;
                    }
                if (canBeColored)
```

```
                {
                    if (i != colored)
                    {
                        var node = array[i];
                        array[i] = array[colored];
                        array[colored] = node;
                    }
                    inDepthColors[0, array[colored] - 1] =
initialColorsNumber;
                    initialColors[array[colored] - 1] =
initialColorsNumber;
                    initialColorClasses[initialColorsNumber
- 1][initialNodesNumInColorClass[initialColorsNumber -
1]] = array[colored];

initialNodesNumInColorClass[initialColorsNumber - 1]++;
                    colored++;
                }
            }
            if (colored == NodesNumber)
                break;
        }
    }

    private int Recolor(int depth)
    {
        int colorsNumber = 0;
        var colorClasses = new int[NodesNumber][];
        var nodesNumInColorClass = new int[NodesNumber];
        skippedNodesNumber[depth] = 0;
        // color vertices, find color classes
        for (int i = 0; i < numberOfNodesArr[depth]; i++)
        {
            int vert = levelNodes[depth, i];
            bool isAdded = false;

            for (int j = 0; j < colorsNumber; j++)
            {
                bool connected = false;
                for (int k = 0; k <
nodesNumInColorClass[j]; k++)
                {
                    if (Graph.Values[vert - 1,
colorClasses[j][k] - 1])
                    {
                        connected = true;
                        break;
                    }
                }
                if (!connected)
                {
```

```
colorClasses[j][nodesNumInColorClass[j]] = vert;
                    inDepthColors[depth, vert - 1] = j + 1;
                    nodesNumInColorClass[j]++;
                    isAdded = true;
                    break;
                }
            }
            if (!isAdded)
            {
                colorClasses[colorsNumber] = new
int[NodesNumber];

colorClasses[colorsNumber][nodesNumInColorClass[colorsNum
ber]] = vert;
                nodesNumInColorClass[colorsNumber]++;
                colorsNumber++;
                inDepthColors[depth, vert - 1] =
colorsNumber;
            }
        }
        return colorsNumber;
    }

    private int RecolorWithSwaps(int depth)
    {
        int colorsNumber = 0;
        int length = numberOfNodesArr[depth];
        skippedNodesNumber[depth] = 0;
        int colored = 0;
        var array = new int[length];
        for (int i = 0; i < length; i++)
        {
            array[i] = levelNodes[depth, i];
        }
        while (true)
        {
            colorsNumber++;
            inDepthColors[depth, array[colored] - 1] =
colorsNumber;
            colored++;
            int lowerBound = colored - 1;
            for (int i = colored; i < length; i++)
            {
                bool canBeColored = true;
                for (int j = lowerBound; j < colored; j++)
                    if (Graph.Values[array[i] - 1, array[j]
- 1])
                    {
                        canBeColored = false;
                        break;
```

```
                        }
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[depth, array[colored] -
     1] = colorsNumber;
                        colored++;
                    }
                }
                if (colored == length)
                    break;
            }
            return colorsNumber;
        }
    }
}
```

Appendix 2

VRecolor-BT-uVP code (VRecolor-BT-uVP.cs)

```
using System.Linq;
using MaximumClique.Base;
using System.Threading.Tasks;
using System.Threading;

namespace MaximumClique.NewAlgorithms
{
    public class VRecolorBtuVP : Algorithm
    {
        private int maxCliqueSize;
        private int[,] levelNodes;
        private int initialColorsNumber;
        private int[][] initialColorClasses;
        private int[] initialNodesNumInColorClass;
        private int[] initialColors;
        private int[,] inDepthColors;
        private int[] numberOfNodesArr;
        private int[] inDepthElementIndex;
        private int[,] skippedNodes;
        private int[] skippedNodesNumber;
        private int[] cache;


        static readonly object key = new object();

        private int ParallDegree;


        public VRecolorBtuVP(Graph graph)
            : base(graph)
        {
            levelNodes = new int[NodesNumber, NodesNumber];
            initialColorClasses = new int[NodesNumber][];
            initialNodesNumInColorClass = new int[NodesNumber];
            initialColors = new int[NodesNumber];
            inDepthColors = new int[NodesNumber, NodesNumber];
            numberOfNodesArr = new int[NodesNumber + 1];
            inDepthElementIndex = new int[NodesNumber + 1];

            skippedNodes = new int[NodesNumber, NodesNumber];
            skippedNodesNumber = new int[NodesNumber + 1];
            cache = new int[NodesNumber];

            ParallDegree = 2;
////////////////////////////////// set for {
MaxDegreeOfParallelism = ParallDegree }
        }
```

```
        public override int Result
        {
            get { return maxCliqueSize; }
        }

        protected override void Solution()
        {
            if (Graph.Density < 0.35)
                InitialColoringWithSwaps();
            else
            { InitialColoring(); }

            int[] inDepthDegree = new int[NodesNumber];


                for (int c = 0; c < initialColorsNumber; c++)
                {
                    skippedNodesNumber[0] = 0;
                    int depth = 0;
                    numberOfNodesArr[depth] = 0;
                    inDepthDegree[depth] = 0;
                    for (int i = 0; i <= c; i++)
                    {
                        for (int j = 0; j <
initialNodesNumInColorClass[i]; j++)
                        {
                            levelNodes[depth,
numberOfNodesArr[depth]] = initialColorClasses[i][j];
                            numberOfNodesArr[depth]++;
                        }
                    }
                    inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;
                    while (depth >= 0)
                    {
                        int inDepthIndex =
inDepthElementIndex[depth];
                        if (inDepthIndex == -1)
                        {
                            depth--;
                            continue;
                        }
                        int p = levelNodes[depth,
inDepthIndex];
                        var color = initialColors[p - 1];
                        if (color < c + 1 && depth +
cache[color - 1] <= maxCliqueSize)
                        {
                            depth--;
                            continue;
```

```
                        }
                        if ((depth + inDepthColors[depth, p -
1]
                            <= maxCliqueSize) &&
CanBeSkipped(inDepthIndex, depth, c + 1))
                        {
                            skippedNodes[depth,
skippedNodesNumber[depth]] = p;
                            skippedNodesNumber[depth]++;
                            inDepthElementIndex[depth]--;
                            continue;
                        }
                        branches++;
                        int prevDepth = depth;
                        depth++;
                        numberOfNodesArr[depth] = 0;
                        inDepthElementIndex[depth] = 0;

                        for (int i = 0; i < inDepthIndex; i++)
                        {
                            if
(Graph.Values[levelNodes[prevDepth, inDepthIndex] - 1,
levelNodes[prevDepth, i] - 1])
                            {
                                levelNodes[depth,
numberOfNodesArr[depth]] = levelNodes[prevDepth, i];
                                numberOfNodesArr[depth]++;
                            }
                        }




                        for (int i =
skippedNodesNumber[prevDepth] - 1; i >= 0; i--)
                        {
                            if
(Graph.Values[levelNodes[prevDepth, inDepthIndex] - 1,
skippedNodes[prevDepth, i] - 1])
                            {
                                levelNodes[depth,
numberOfNodesArr[depth]] = skippedNodes[prevDepth, i];
                                numberOfNodesArr[depth]++;
                            }
                        }

                        inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;

                        if (numberOfNodesArr[depth] > 0)
                        {
```

```
                            int colNum = Graph.Density < 0.55 ?
RecolorWithSwaps(depth) : Recolor(depth);

                            if (depth + colNum <=
maxCliqueSize)
                                depth--;
                        }
                        else
                        {
                            if (depth > maxCliqueSize)
                            {
                                maxCliqueSize = depth;
                                break;
                            }
                            depth--;
                        }
                        inDepthElementIndex[prevDepth]--;

                    }
                    cache[c] = maxCliqueSize;

                }
            }

            private bool CanBeSkipped(int vertIndex, int depth, int
currentColor)
            {
                int threshold = maxCliqueSize - depth;
                int vert = levelNodes[depth, vertIndex];

                // on current depth on what vertex index we will
cut??
                // if (color < c + 1 && depth + cache[color - 1] <=
maxCliqueSize){ depth--; }
                int initialColorThresholdIndex = -1;
                for (int i = vertIndex - 1; i >= 0; i--)
                {
                    int vert2 = levelNodes[depth, i];
                    if (initialColorThresholdIndex == -1)
                    {
                        int color = initialColors[vert2 - 1];
                        if (color < currentColor && cache[color -
1] <= threshold)
                        {
                            initialColorThresholdIndex = i;
                            break;
                        }
                    }
                }
```

```
            // if we want to skip a vertex, we have to check if
current vertex
            // is adjacent to any vertex INSIDE vertices that
will be CUT!! (from initialColorThresholdIndex to zero) on this
depth
            // with color higher than threshold
            for (int i = initialColorThresholdIndex; i >= 0;
i--)
            {
                int vert2 = levelNodes[depth, i];
                if (Graph.Values[vert - 1, vert2 - 1])
                {
                    if (inDepthColors[depth, vert2 - 1] >
threshold)
                        return false;
                }
            }
            return true;
        }

        public int FindNumberOfColorClasses(int depth, int
numberOfNodes)
        {
            int nPrevColor = 0;

            int numberOfColorClasses = 0;
            //for (int i = 0; i < numberOfNodes; i++)
            Parallel.For(0, numberOfNodes, new ParallelOptions
{ MaxDegreeOfParallelism = ParallDegree }, i=>
            {
                int currentColor =
initialColors[levelNodes[depth, i] - 1];
                if (currentColor != nPrevColor)
                {
                    numberOfColorClasses++; nPrevColor =
currentColor;
                }
            });
            return numberOfColorClasses;
        }

        private void InitialColoring()
        {
            var verticesWithDegrees = new int[NodesNumber][];
//var verticesWithDegrees = new int[NodesNumber][];
            // count degrees of vertices

            Parallel.For(0, NodesNumber, new ParallelOptions {
MaxDegreeOfParallelism = ParallDegree }, i =>
/////////////////////////// Parallel.For
            {
```

```
                verticesWithDegrees[i] = new int[2];
                verticesWithDegrees[i][0] = i + 1;
            });

            Parallel.For(0, NodesNumber, new ParallelOptions {
MaxDegreeOfParallelism = ParallDegree }, i =>
//////////////////////////// Parallel.For
            {
                for (int j = i + 1; j < NodesNumber; j++)
                    if (Graph.Values[i, j])
                    {
                        verticesWithDegrees[i][1]++;
                        verticesWithDegrees[j][1]++;
                    }
            });

            // order vertices by degree
            var orderedVertices =
verticesWithDegrees.OrderByDescending(i => i[1]).ToArray();

            // color vertices, find color classes
            //Parallel.For(0, NodesNumber, i =>
            //{
            for (int i = 0; i < NodesNumber; i++)
                {
                    int vert = orderedVertices[i][0];
                    bool isAdded = false;
                    //Parallel.For(0, initialColorsNumber, (j,
loopStateJ )=>
                    for (int j = 0; j < initialColorsNumber;
j++)
                    {
                        bool connected = false;
                        for (int k = 0; k <
initialNodesNumInColorClass[j]; k++)
                        {
                            if (Graph.Values[vert - 1,
initialColorClasses[j][k] - 1])
                            {
                                connected = true;
                                break;
                            }
                        }
                        if (!connected)
                        {

initialColorClasses[j][initialNodesNumInColorClass[j]] = vert;
                            initialColors[vert - 1] = j + 1;
                            inDepthColors[0, vert - 1] = j + 1;
                            initialNodesNumInColorClass[j]++;
                            isAdded = true;
```

```
                            break;
                            //loopStateJ.Break();

                        }
                    }//);
                    if (!isAdded)
                    {

initialColorClasses[initialColorsNumber] = new
int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInColor
Class[initialColorsNumber]] = vert;

initialNodesNumInColorClass[initialColorsNumber]++;
                        initialColorsNumber++;
                        initialColors[vert - 1] =
initialColorsNumber;
                        inDepthColors[0, vert - 1] =
initialColorsNumber;
                    }
            }
            //});
        }

        private void InitialColoringWithSwaps()
        {
            var array = new int[NodesNumber];

            Parallel.For(0, NodesNumber, new ParallelOptions {
MaxDegreeOfParallelism = ParallDegree }, i => array[i] = i +
1);                    ////////////////////////////
Parallel.For


            int colored = 0;
            initialColorsNumber = 0;

            while (true)
            {
                initialColorClasses[initialColorsNumber] = new
int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInColor
Class[initialColorsNumber]] = array[colored];

initialNodesNumInColorClass[initialColorsNumber]++;
                initialColorsNumber++;
                initialColors[array[colored] - 1] =
initialColorsNumber;
```

```
                inDepthColors[0, array[colored] - 1] =
initialColorsNumber;

                colored++;
                int lowerBound = colored - 1;
                //Parallel.For(colored, NodesNumber, i =>
                for (int i = colored; i < NodesNumber; i++)
                {
                    bool canBeColored = true;
                    //Parallel.For(lowerBound, colored, (j,
loopStateJ) =>
                    //{
                    for (int j = lowerBound; j < colored; j++)
                    if (Graph.Values[array[i] - 1, array[j] -
1])
                        {
                        canBeColored = false;
                        break;
                        //loopStateJ.Break();
                        }
                    //});
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[0, array[colored] - 1] =
initialColorsNumber;
                        initialColors[array[colored] - 1] =
initialColorsNumber;
                        initialColorClasses[initialColorsNumber
- 1][initialNodesNumInColorClass[initialColorsNumber - 1]] =
array[colored];

initialNodesNumInColorClass[initialColorsNumber - 1]++;
                        colored++;
                    }
                }//);
                if (colored == NodesNumber)
                    break;
            }
        }

        private int Recolor(int depth)
        {
            int colorsNumber = 0;
            var colorClasses = new int[NodesNumber][];
            var nodesNumInColorClass = new int[NodesNumber];
```

```
            skippedNodesNumber[depth] = 0;
            // color vertices, find color classes

            for (int i = 0; i < numberOfNodesArr[depth]; i++)
            {

                    int vert = levelNodes[depth, i];
                    bool isAdded = false;

                    for (int j = 0; j < colorsNumber; j++)
                    {
                        bool connected = false;
                        for (int k = 0; k <
nodesNumInColorClass[j]; k++)
                        {
                            if (Graph.Values[vert - 1,
colorClasses[j][k] - 1])
                            {
                                connected = true;
                                break;
                            }
                        }


                        if (!connected)
                        {

colorClasses[j][nodesNumInColorClass[j]] = vert;
                            inDepthColors[depth, vert - 1] = j
+ 1;
                            nodesNumInColorClass[j]++;
                            isAdded = true;
                            break;
                        }
                    }


                    if (!isAdded)
                    {
                        colorClasses[colorsNumber] = new
int[NodesNumber];

colorClasses[colorsNumber][nodesNumInColorClass[colorsNumber]]
= vert;
                        nodesNumInColorClass[colorsNumber]++;
                        colorsNumber++;
                        inDepthColors[depth, vert - 1] =
colorsNumber;
                    }
            }
            return colorsNumber;
```

```
        }


        private int RecolorWithSwaps(int depth)
        {
            int colorsNumber = 0;
            int length = numberOfNodesArr[depth];
            skippedNodesNumber[depth] = 0;
            int colored = 0;
            var array = new int[length];

            Parallel.For(0, length, new ParallelOptions {
MaxDegreeOfParallelism = ParallDegree }, i =>
            {
                array[i] = levelNodes[depth, i];
            });

            while (true)
            {
                colorsNumber++;
                inDepthColors[depth, array[colored] - 1] =
colorsNumber;
                colored++;
                int lowerBound = colored - 1;
                for (int i = colored; i < length; i++)
                {
                    bool canBeColored = true;
                    for (int j = lowerBound; j < colored; j++)
                        if (Graph.Values[array[i] - 1, array[j]
- 1])
                        {
                            canBeColored = false;
                            break;
                        }
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[depth, array[colored] -
1] = colorsNumber;
                        colored++;
                    }
                }
                if (colored == length)
                    break;
            }
            return colorsNumber;
```

```
            }
        }
    }
```

Appendix 3

VRecolor-BT-uVP2 code (VRecolor-BT-uVP2.cs)

```
using System.Linq;
using MaximumClique.Base;
using System.Threading.Tasks;
using System.Threading;

namespace MaximumClique.NewAlgorithms
{
    public class VRecolorBtuVP2 : Algorithm
    {
        private int maxCliqueSize;
        private int[,] levelNodes;
        private int initialColorsNumber;
        private int[][] initialColorClasses;
        private int[] initialNodesNumInColorClass;
        private int[] initialColors;
        private int[,] inDepthColors;
        private int[] numberOfNodesArr;
        private int[] inDepthElementIndex;
        private int[,] skippedNodes;
        private int[] skippedNodesNumber;
        private int[] cache;

        public VRecolorBtuVP2(Graph graph)
            : base(graph)
        {
            levelNodes = new int[NodesNumber, NodesNumber];
            initialColorClasses = new int[NodesNumber][];
            initialNodesNumInColorClass = new int[NodesNumber];
            initialColors = new int[NodesNumber];
            inDepthColors = new int[NodesNumber, NodesNumber];
            numberOfNodesArr = new int[NodesNumber + 1];
            inDepthElementIndex = new int[NodesNumber + 1];

            skippedNodes = new int[NodesNumber, NodesNumber];
            skippedNodesNumber = new int[NodesNumber + 1];
            cache = new int[NodesNumber];
        }

        public override int Result
        {
            get { return maxCliqueSize; }
        }

        protected override void Solution()
        {
            if (Graph.Density < 0.35)
```

```
                InitialColoringWithSwaps();
            else
                InitialColoring();

            int[] inDepthDegree = new int[NodesNumber];

            for (int c = 0; c < initialColorsNumber; c++)
            {
                skippedNodesNumber[0] = 0;
                int depth = 0;
                numberOfNodesArr[depth] = 0;
                inDepthDegree[depth] = 0;

                Thread[] threads = new
Thread[initialColorsNumber];
                Thread[] threadss = new
Thread[initialColorsNumber];

                threads[c] = new Thread(NILL =>
                {
                    SolutionFoFor(c, depth);
                });

                threads[c].Priority = ThreadPriority.Highest;
                threads[c].Start();
                threads[c].Join();

                inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;

                threadss[c] = new Thread(NILL =>
                {

                    depth = SolutionWhile(c, depth);

                });

                threadss[c].Priority = ThreadPriority.Highest;
                threadss[c].Start();
                threadss[c].Join();


                cache[c] = maxCliqueSize;
            }
        }

        private int SolutionWhile(int c, int depth)
        {
            while (depth >= 0)
            {
                int inDepthIndex = inDepthElementIndex[depth];
```

```
                if (inDepthIndex == -1)
                {
                    depth--;
                    continue;
                }
                int p = levelNodes[depth, inDepthIndex];
                var color = initialColors[p - 1];
                if (color < c + 1 && depth + cache[color - 1]
<= maxCliqueSize)
                {
                    depth--;
                    continue;
                }
                if ((depth + inDepthColors[depth, p - 1]
                    <= maxCliqueSize) &&
CanBeSkipped(inDepthIndex, depth, c + 1))
                {
                    skippedNodes[depth,
skippedNodesNumber[depth]] = p;
                    skippedNodesNumber[depth]++;
                    inDepthElementIndex[depth]--;
                    continue;
                }
                branches++;
                int prevDepth = depth;
                depth++;
                numberOfNodesArr[depth] = 0;
                inDepthElementIndex[depth] = 0;

                for (int i = 0; i < inDepthIndex; i++)
                {
                    if (Graph.Values[levelNodes[prevDepth,
inDepthIndex] - 1, levelNodes[prevDepth, i] - 1])
                    {
                        levelNodes[depth,
numberOfNodesArr[depth]] = levelNodes[prevDepth, i];
                        numberOfNodesArr[depth]++;
                    }
                }
                for (int i = skippedNodesNumber[prevDepth] - 1;
i >= 0; i--)
                {
                    if (Graph.Values[levelNodes[prevDepth,
inDepthIndex] - 1, skippedNodes[prevDepth, i] - 1])
                    {
                        levelNodes[depth,
numberOfNodesArr[depth]] = skippedNodes[prevDepth, i];
                        numberOfNodesArr[depth]++;
                    }
                }
```

```
                inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;

                if (numberOfNodesArr[depth] > 0)
                {
                    int colNum = Graph.Density < 0.55 ?
RecolorWithSwaps(depth) : Recolor(depth);

                    if (depth + colNum <= maxCliqueSize)
                        depth--;
                }
                else
                {
                    if (depth > maxCliqueSize)
                    {
                        maxCliqueSize = depth;
                        break;
                    }
                    depth--;
                }
                inDepthElementIndex[prevDepth]--;

            }

            return depth;
        }

        private void SolutionFoFor(int c, int depth)
        {
            for (int i = 0; i <= c; i++)
            {
                for (int j = 0; j <
initialNodesNumInColorClass[i]; j++)
                {
                    levelNodes[depth, numberOfNodesArr[depth]]
= initialColorClasses[i][j];
                    numberOfNodesArr[depth]++;
                }
            }
        }

        private bool CanBeSkipped(int vertIndex, int depth, int
currentColor)
        {
            int threshold = maxCliqueSize - depth;
            int vert = levelNodes[depth, vertIndex];

            // on current depth on what vertex index we will
cut??
            // if (color < c + 1 && depth + cache[color - 1] <=
maxCliqueSize){ depth--; }
```

```
            int initialColorThresholdIndex = -1;
            for (int i = vertIndex - 1; i >= 0; i--)
            {
                int vert2 = levelNodes[depth, i];
                if (initialColorThresholdIndex == -1)
                {
                    int color = initialColors[vert2 - 1];
                    if (color < currentColor && cache[color -
1] <= threshold)
                    {
                        initialColorThresholdIndex = i;
                        break;
                    }
                }
            }

            // if we want to skip a vertex, we have to check if
current vertex
            // is adjacent to any vertex INSIDE vertices that
will be CUT!! (from initialColorThresholdIndex to zero) on this
depth
            // with color higher than threshold
            for (int i = initialColorThresholdIndex; i >= 0;
i--)
            {
                int vert2 = levelNodes[depth, i];
                if (Graph.Values[vert - 1, vert2 - 1])
                {
                    if (inDepthColors[depth, vert2 - 1] >
threshold)
                        return false;
                }
            }
            return true;
        }

        public int FindNumberOfColorClasses(int depth, int
numberOfNodes)
        {
            int nPrevColor = 0;

            int numberOfColorClasses = 0;
            for (int i = 0; i < numberOfNodes; i++)
            {
                int currentColor =
initialColors[levelNodes[depth, i] - 1];
                if (currentColor != nPrevColor)
                {
                    numberOfColorClasses++; nPrevColor =
currentColor;
                }
```

```
            }
            return numberOfColorClasses;
        }

        private void InitialColoring()
        {
            var verticesWithDegrees = new int[NodesNumber][];
//var verticesWithDegrees = new int[NodesNumber][];
            // count degrees of vertices

            for (int i = 0; i < NodesNumber; i++)
            {
                verticesWithDegrees[i] = new int[2];
                verticesWithDegrees[i][0] = i + 1;
            }

            for (int i = 0; i < NodesNumber; i++)
            {
                for (int j = i + 1; j < NodesNumber; j++)
                    if (Graph.Values[i, j])
                    {
                        verticesWithDegrees[i][1]++;
                        verticesWithDegrees[j][1]++;
                    }
            }

            // order vertices by degree
            var orderedVertices =
verticesWithDegrees.OrderByDescending(i => i[1]).ToArray();


            for (int i = 0; i < NodesNumber; i++)
            {
                int vert = orderedVertices[i][0];
                bool isAdded = false;

                for (int j = 0; j < initialColorsNumber; j++)
                {
                    bool connected = false;
                    for (int k = 0; k <
initialNodesNumInColorClass[j]; k++)
                    {
                        if (Graph.Values[vert - 1,
initialColorClasses[j][k] - 1])
                        {
                            connected = true;
                            break;
                        }
                    }
                    if (!connected)
                    {
```

```
initialColorClasses[j][initialNodesNumInColorClass[j]] = vert;
                        initialColors[vert - 1] = j + 1;
                        inDepthColors[0, vert - 1] = j + 1;
                        initialNodesNumInColorClass[j]++;
                        isAdded = true;
                        break;


                    }
                }
                if (!isAdded)
                {
                    initialColorClasses[initialColorsNumber] =
new int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInColor
Class[initialColorsNumber]] = vert;

initialNodesNumInColorClass[initialColorsNumber]++;
                    initialColorsNumber++;
                    initialColors[vert - 1] =
initialColorsNumber;
                    inDepthColors[0, vert - 1] =
initialColorsNumber;
                }
            }

        }

        private void InitialColoringWithSwaps()
        {
            var array = new int[NodesNumber];

            for (int i = 0; i < NodesNumber; i++)
                array[i] = i + 1;

            int colored = 0;
            initialColorsNumber = 0;

            while (true)
            {
                initialColorClasses[initialColorsNumber] = new
int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInColor
Class[initialColorsNumber]] = array[colored];

initialNodesNumInColorClass[initialColorsNumber]++;
                initialColorsNumber++;
```

```
                initialColors[array[colored] - 1] =
initialColorsNumber;
                inDepthColors[0, array[colored] - 1] =
initialColorsNumber;

                colored++;
                int lowerBound = colored - 1;

                for (int i = colored; i < NodesNumber; i++)
                {
                    bool canBeColored = true;

                    for (int j = lowerBound; j < colored; j++)
                        if (Graph.Values[array[i] - 1, array[j]
- 1])
                        {
                            canBeColored = false;
                            break;

                        }

                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[0, array[colored] - 1] =
initialColorsNumber;
                        initialColors[array[colored] - 1] =
initialColorsNumber;
                        initialColorClasses[initialColorsNumber
- 1][initialNodesNumInColorClass[initialColorsNumber - 1]] =
array[colored];

initialNodesNumInColorClass[initialColorsNumber - 1]++;
                        colored++;
                    }
                }
                if (colored == NodesNumber)
                    break;
            }
        }

        private int Recolor(int depth)
        {
            int colorsNumber = 0;
            var colorClasses = new int[NodesNumber][];
            var nodesNumInColorClass = new int[NodesNumber];
```

```
            skippedNodesNumber[depth] = 0;
            // color vertices, find color classes
            for (int i = 0; i < numberOfNodesArr[depth]; i++)
            {
                int vert = levelNodes[depth, i];
                bool isAdded = false;

                for (int j = 0; j < colorsNumber; j++)
                {
                    bool connected = false;
                    for (int k = 0; k <
nodesNumInColorClass[j]; k++)
                    {
                        if (Graph.Values[vert - 1,
colorClasses[j][k] - 1])
                        {
                            connected = true;
                            break;
                        }
                    }
                    if (!connected)
                    {

colorClasses[j][nodesNumInColorClass[j]] = vert;
                        inDepthColors[depth, vert - 1] = j + 1;
                        nodesNumInColorClass[j]++;
                        isAdded = true;
                        break;

                    }
                }
                if (!isAdded)
                {
                    colorClasses[colorsNumber] = new
int[NodesNumber];

colorClasses[colorsNumber][nodesNumInColorClass[colorsNumber]]
= vert;
                    nodesNumInColorClass[colorsNumber]++;
                    colorsNumber++;
                    inDepthColors[depth, vert - 1] =
colorsNumber;
                }
            }
            return colorsNumber;
        }

        private int RecolorWithSwaps(int depth)
        {
            int colorsNumber = 0;
            int length = numberOfNodesArr[depth];
```

```
            skippedNodesNumber[depth] = 0;
            int colored = 0;
            var array = new int[length];

            for (int i = 0; i < length; i++)
            {
                array[i] = levelNodes[depth, i];
            }

            while (true)
            {
                colorsNumber++;
                inDepthColors[depth, array[colored] - 1] =
colorsNumber;
                colored++;
                int lowerBound = colored - 1;
                for (int i = colored; i < length; i++)
                {
                    bool canBeColored = true;
                    for (int j = lowerBound; j < colored; j++)
                        if (Graph.Values[array[i] - 1, array[j]
- 1])
                        {
                            canBeColored = false;
                            break;
                        }
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[depth, array[colored] -
1] = colorsNumber;
                        colored++;
                    }
                }
                if (colored == length)
                    break;
            }
            return colorsNumber;
        }
    }
}
```

Appendix 4

VRecolor-BT-uVPtest code (VRecolor-BT-uVPtest.cs)

```
using System.Linq;
using MaximumClique.Base;
using System.Threading.Tasks;
using System.Threading;

namespace MaximumClique.NewAlgorithms
{
    public class VRecolorBtuVPtest : Algorithm
    {
        private int maxCliqueSize;
        private int[,] levelNodes;
        private int initialColorsNumber;
        private int[][] initialColorClasses;
        private int[] initialNodesNumInColorClass;
        private int[] initialColors;
        private int[,] inDepthColors;
        private int[] numberOfNodesArr;
        private int[] inDepthElementIndex;
        private int[,] skippedNodes;
        private int[] skippedNodesNumber;
        private int[] cache;


        static readonly object key = new object();
        static readonly object key2 = new object();


        public VRecolorBtuVPtest(Graph graph)
            : base(graph)
        {
            levelNodes = new int[NodesNumber, NodesNumber];
            initialColorClasses = new int[NodesNumber][];
            initialNodesNumInColorClass = new
int[NodesNumber];
            initialColors = new int[NodesNumber];
            inDepthColors = new int[NodesNumber,
NodesNumber];
            numberOfNodesArr = new int[NodesNumber + 1];
            inDepthElementIndex = new int[NodesNumber + 1];

            skippedNodes = new int[NodesNumber,
NodesNumber];
            skippedNodesNumber = new int[NodesNumber + 1];
            cache = new int[NodesNumber];
```

```
        }

        public override int Result
        {
            get { return maxCliqueSize; }
        }

        protected override void Solution()
        {
            if (Graph.Density < 0.35)
                InitialColoringWithSwaps();
            else
            { InitialColoring(); }

            int[] inDepthDegree = new int[NodesNumber];


            //Parallel.For(0, initialColorsNumber, new
ParallelOptions { MaxDegreeOfParallelism = 2}, c =>
            //{
            //    lock (key)
            for (int c = 0; c < initialColorsNumber; c++)
            {
                skippedNodesNumber[0] = 0;
                int depth = 0;
                numberOfNodesArr[depth] = 0;
                inDepthDegree[depth] = 0;
                for (int i = 0; i <= c; i++)
                {
                    for (int j = 0; j <
initialNodesNumInColorClass[i]; j++)
                    {
                        levelNodes[depth,
numberOfNodesArr[depth]] = initialColorClasses[i][j];
                        numberOfNodesArr[depth]++;
                    }
                }
                inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;
                while (depth >= 0)
                {
                    int inDepthIndex =
inDepthElementIndex[depth];
                    if (inDepthIndex == -1)
                    {
                        depth--;
                        continue;
                    }
```

```
                    int p = levelNodes[depth,
inDepthIndex];
                    var color = initialColors[p - 1];
                    if (color < c + 1 && depth +
cache[color - 1] <= maxCliqueSize)
                    {
                        depth--;
                        continue;
                    }
                    if ((depth + inDepthColors[depth, p -
1]
                        <= maxCliqueSize) &&
CanBeSkipped(inDepthIndex, depth, c + 1))
                    {
                        skippedNodes[depth,
skippedNodesNumber[depth]] = p;
                        skippedNodesNumber[depth]++;
                        inDepthElementIndex[depth]--;
                        continue;
                    }
                    branches++;
                    int prevDepth = depth;
                    depth++;
                    numberOfNodesArr[depth] = 0;
                    inDepthElementIndex[depth] = 0;

                    //for (int i = 0; i < inDepthIndex;
i++)
                    //{
                    //    if
(Graph.Values[levelNodes[prevDepth, inDepthIndex] - 1,
levelNodes[prevDepth, i] - 1])
                    //    {
                    //        levelNodes[depth,
numberOfNodesArr[depth]] = levelNodes[prevDepth, i];
                    //        numberOfNodesArr[depth]++;
                    //    }
                    //}


                    Parallel.For(0, inDepthIndex, new
ParallelOptions { MaxDegreeOfParallelism = 5 }, i =>
                    {
                        if
(Graph.Values[levelNodes[prevDepth, inDepthIndex] - 1,
levelNodes[prevDepth, i] - 1])
                        {

                            lock (key)
```

```
                            {
                                levelNodes[depth,
numberOfNodesArr[depth]] = levelNodes[prevDepth, i];
                                numberOfNodesArr[depth]++;
                            }


                        }

                    });

                    //    for (int i =
skippedNodesNumber[prevDepth] - 1; i >= 0; i--)
                    //{
                    //    if
(Graph.Values[levelNodes[prevDepth, inDepthIndex] - 1,
skippedNodes[prevDepth, i] - 1])
                    //    {
                    //        levelNodes[depth,
numberOfNodesArr[depth]] = skippedNodes[prevDepth, i];
                    //        numberOfNodesArr[depth]++;
                    //    }
                    //}

                    inDepthElementIndex[depth] =
numberOfNodesArr[depth] - 1;

                    if (numberOfNodesArr[depth] > 0)
                    {
                        int colNum = Graph.Density < 0.55 ?
RecolorWithSwaps(depth) : Recolor(depth);

                        if (depth + colNum <=
maxCliqueSize)
                            depth--;
                    }
                    else
                    {
                        if (depth > maxCliqueSize)
                        {
                            maxCliqueSize = depth;
                            break;
                        }
                        depth--;
                    }
                    inDepthElementIndex[prevDepth]--;

                }
                cache[c] = maxCliqueSize;
```

```
                // }
            }//);
        }

        private bool CanBeSkipped(int vertIndex, int depth,
int currentColor)
        {
            int threshold = maxCliqueSize - depth;
            int vert = levelNodes[depth, vertIndex];

            // on current depth on what vertex index we
will cut??
            // if (color < c + 1 && depth + cache[color -
1] <= maxCliqueSize){ depth--; }
            int initialColorThresholdIndex = -1;
            for (int i = vertIndex - 1; i >= 0; i--)
            {
                int vert2 = levelNodes[depth, i];
                if (initialColorThresholdIndex == -1)
                {
                    int color = initialColors[vert2 - 1];
                    if (color < currentColor && cache[color
- 1] <= threshold)
                    {
                        initialColorThresholdIndex = i;
                        break;
                    }
                }
            }

            // if we want to skip a vertex, we have to
check if current vertex
            // is adjacent to any vertex INSIDE vertices
that will be CUT!! (from initialColorThresholdIndex to
zero) on this depth
            // with color higher than threshold
            for (int i = initialColorThresholdIndex; i >=
0; i--)
            {
                int vert2 = levelNodes[depth, i];
                if (Graph.Values[vert - 1, vert2 - 1])
                {
                    if (inDepthColors[depth, vert2 - 1] >
threshold)
                        return false;
                }
            }
            return true;
        }
```

```
        public int FindNumberOfColorClasses(int depth, int
numberOfNodes)
        {
            int nPrevColor = 0;

            int numberOfColorClasses = 0;
            for (int i = 0; i < numberOfNodes; i++)
            {
                int currentColor =
initialColors[levelNodes[depth, i] - 1];
                if (currentColor != nPrevColor)
                {
                    numberOfColorClasses++; nPrevColor =
currentColor;
                }
            }
            return numberOfColorClasses;
        }

        private void InitialColoring()
        {
            var verticesWithDegrees = new
int[NodesNumber][]; //var verticesWithDegrees = new
int[NodesNumber][];
            // count degrees of vertices

            //Parallel.For(0, NodesNumber, i =>
            for (int i = 0; i < NodesNumber; i++)
            {
                verticesWithDegrees[i] = new int[2];
                verticesWithDegrees[i][0] = i + 1;
            }//);

            //Parallel.For(0, NodesNumber, i =>
            //{
            for (int i = 0; i < NodesNumber; i++)
                for (int j = i + 1; j < NodesNumber; j++)
                    if (Graph.Values[i, j])
                    {
                        verticesWithDegrees[i][1]++;
                        verticesWithDegrees[j][1]++;
                    }
            //});

            // order vertices by degree
            var orderedVertices =
verticesWithDegrees.OrderByDescending(i => i[1]).ToArray();
```

```
            // color vertices, find color classes
            //Parallel.For(0, NodesNumber, i =>
            //{
            for (int i = 0; i < NodesNumber; i++)
            {
                int vert = orderedVertices[i][0];
                bool isAdded = false;
                //Parallel.For(0, initialColorsNumber, (j,
loopStateJ )=>
                for (int j = 0; j < initialColorsNumber;
j++)
                {
                    bool connected = false;
                    for (int k = 0; k <
initialNodesNumInColorClass[j]; k++)
                    {
                        if (Graph.Values[vert - 1,
initialColorClasses[j][k] - 1])
                        {
                            connected = true;
                            break;
                        }
                    }
                    if (!connected)
                    {

initialColorClasses[j][initialNodesNumInColorClass[j]] =
vert;
                        initialColors[vert - 1] = j + 1;
                        inDepthColors[0, vert - 1] = j + 1;
                        initialNodesNumInColorClass[j]++;
                        isAdded = true;
                        break;
                        //loopStateJ.Break();

                    }
                }//);
                if (!isAdded)
                {

initialColorClasses[initialColorsNumber] = new
int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInC
olorClass[initialColorsNumber]] = vert;

initialNodesNumInColorClass[initialColorsNumber]++;
                    initialColorsNumber++;
```

```
                    initialColors[vert - 1] =
initialColorsNumber;
                    inDepthColors[0, vert - 1] =
initialColorsNumber;
                }
            }
            //});
        }

        private void InitialColoringWithSwaps()
        {
            var array = new int[NodesNumber];
            for (int i = 0; i < NodesNumber; i++)
                array[i] = i + 1;
            //Parallel.For(0, NodesNumber, i => array[i] =
i + 1);

            int colored = 0;
            initialColorsNumber = 0;

            while (true)
            {
                initialColorClasses[initialColorsNumber] =
new int[NodesNumber];

initialColorClasses[initialColorsNumber][initialNodesNumInC
olorClass[initialColorsNumber]] = array[colored];

initialNodesNumInColorClass[initialColorsNumber]++;
                initialColorsNumber++;
                initialColors[array[colored] - 1] =
initialColorsNumber;
                inDepthColors[0, array[colored] - 1] =
initialColorsNumber;

                colored++;
                int lowerBound = colored - 1;
                //Parallel.For(colored, NodesNumber, i =>
                for (int i = colored; i < NodesNumber; i++)
                {
                    bool canBeColored = true;
                    //Parallel.For(lowerBound, colored, (j,
loopStateJ) =>
                    //{
                    for (int j = lowerBound; j < colored;
j++)
                        if (Graph.Values[array[i] - 1,
array[j] - 1])
                        {
```

```
                            canBeColored = false;
                            break;
                            //loopStateJ.Break();
                        }
                    //});
                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[0, array[colored] -
1] = initialColorsNumber;
                        initialColors[array[colored] - 1] =
initialColorsNumber;

initialColorClasses[initialColorsNumber -
1][initialNodesNumInColorClass[initialColorsNumber - 1]] =
array[colored];

initialNodesNumInColorClass[initialColorsNumber - 1]++;
                        colored++;
                    }
                }//);
                if (colored == NodesNumber)
                    break;
            }
        }

        private int Recolor(int depth)
        {
            int colorsNumber = 0;
            var colorClasses = new int[NodesNumber][];
            var nodesNumInColorClass = new
int[NodesNumber];
            skippedNodesNumber[depth] = 0;
            // color vertices, find color classes


            //Parallel.For(0, numberOfNodesArr[depth], new
ParallelOptions { MaxDegreeOfParallelism = 1 }, i =>
            for (int i = 0; i < numberOfNodesArr[depth];
i++)
            {
                //lock (key2)
                //{
                int vert = levelNodes[depth, i];
```

```
                bool isAdded = false;

                ////////////////////// old code VRecolorBtu
//////////////////////


                //for (int j = 0; j < colorsNumber; j++)
                //{
                //    bool connected = false;
                //    for (int k = 0; k <
nodesNumInColorClass[j]; k++)
                //    {
                //        if (Graph.Values[vert - 1,
colorClasses[j][k] - 1]) /////// old code
                //        {
                //            connected = true;
                //            break;
                //        }
                //    }


                ////////////////// new code
VRecolorBtuVPtest //////////////////////
                for (int j = 0; j < colorsNumber; j++)
                {
                    bool connected = false;
                    var resRecolor = Parallel.For(0,
nodesNumInColorClass[j], new ParallelOptions {
MaxDegreeOfParallelism = 2 }, (k, loopStateK) =>
                    {
                        if (Graph.Values[vert - 1,
colorClasses[j][k] - 1])
                        {
                            //lock (key2)
                            //{
                            //    connected = true;
                            //    //break;;
                            //}
                            loopStateK.Break();

                        }
                    });

                    if (!resRecolor.IsCompleted)
                    {
                        connected = true;
                    }
```

```
                    if (!connected)
                    {

colorClasses[j][nodesNumInColorClass[j]] = vert;
                        inDepthColors[depth, vert - 1] = j
+ 1;
                        nodesNumInColorClass[j]++;
                        isAdded = true;
                        break;
                    }
                }


                if (!isAdded)
                {
                    colorClasses[colorsNumber] = new
int[NodesNumber];

colorClasses[colorsNumber][nodesNumInColorClass[colorsNumbe
r]] = vert;
                    nodesNumInColorClass[colorsNumber]++;
                    colorsNumber++;
                    inDepthColors[depth, vert - 1] =
colorsNumber;
                }

            }
            return colorsNumber;
        }


        private int RecolorWithSwaps(int depth)
        {
            int colorsNumber = 0;
            int length = numberOfNodesArr[depth];
            skippedNodesNumber[depth] = 0;
            int colored = 0;
            var array = new int[length];
            for (int i = 0; i < length; i++)
            {
                array[i] = levelNodes[depth, i];
            }
            while (true)
            {
                colorsNumber++;
                inDepthColors[depth, array[colored] - 1] =
colorsNumber;
                colored++;
                int lowerBound = colored - 1;
```

```
                for (int i = colored; i < length; i++)
                {

                    ////////////////////////// old code
//////////////////////

                    //bool canBeColored = true;
                    //for (int j = lowerBound; j < colored;
j++)
                    //    if (Graph.Values[array[i] - 1,
array[j] - 1])
                    //    {
                    //        canBeColored = false;
                    //        break;
                    //    }

                    bool canBeColored = true;
                    var resW = Parallel.For(lowerBound,
colored, new ParallelOptions { MaxDegreeOfParallelism = 2
}, ( j, loopStateJ) =>
                        {
                        if (Graph.Values[array[i] - 1,
array[j] - 1])
                            {
                                loopStateJ.Break();
                            }
                        });

                    if (!resW.IsCompleted)
                    {
                        canBeColored = false;
                    }

                    if (canBeColored)
                    {
                        if (i != colored)
                        {
                            var node = array[i];
                            array[i] = array[colored];
                            array[colored] = node;
                        }
                        inDepthColors[depth, array[colored]
- 1] = colorsNumber;
                        colored++;
                    }
                }
                if (colored == length)
                    break;
            }
```

```
                return colorsNumber;
            }
        }
    }
```